RAFAEL STEIL

Agradecimentos

Àqueles que, antes de todos, em mim acreditaram: Daniel, Everson, Guilherme e Paulo.

Sumário

Capítulo 1

Introdução

O iPhone e o iPad são fenômenos.

Não só seus marketshares possuem números impressionantes, mas também o número de vendas da App Store traz ânimo para nós, desenvolvedores.

Mesmo com o número expressivo de dispositivos Android, é estimado que a App Store seja responsável por 85-90% do faturamento de todas os aplicativos móveis já vendidos (http://bit.ly/appstoreNumeros) , tendo revertido quase 4 bilhões de dólares aos desenvolvedores de iPad e iPhone.

E o Brasil? De acordo com o site de análise mobile flurry.com, o mercado brasileiro atingiu, em 2012, a décima posição mundial em número de smartphones. São mais de 13 milhões de dispositivos, em sua grande maioria iPhones e Androids. Este site do Google pode agregar mais informações:

http://www.thinkwithgoogle.com/mobileplanet/pt-br/

Você encontra mais sobre o marketshare de dispositivos móveis através dos sites: http://www.netmarketshare.com/ http://gs.statcounter.com/

Você pode estar lendo este livro para criar sua própria aplicação. Mas talvez esteja

procurando um novo emprego, para trabalhar com tecnologias novas. Esse também é um grande mercado, com um número de vagas crescendo.

Para estar nesse mercado, seja com um emprego novo ou para desenvolver sua própria aplicação e colocá-la na App Store, você precisa saber programar para o sistema operacional iOS.

1.1 Desenvolvendo para o iOS

Em 2007, com o lançamento do iPhone, o sistema operacional que roda dentro do dispositivo tinha o criativo nome de iPhone OS. Com a evolução dos dispositivos e a chegada do iPad, o sistema mudou de nome para iOS. O iOS nasceu do já antigo sistema operacional, o OS X.

Curiosamente, no início não havia como desenvolver para o iPhone. O kit de desenvolvimento, o SDK, só ficou disponível em 2008. Nessa época ele era pago, tornando-se gratuito em 2010 (apesar disso, os betas só podem ser utilizados pelos desenvolvedores que possuem uma conta paga na Apple).

Para instalar o SDK e programar para o iOS, você vai necessariamente precisar de um computador que rode o OS X. Basicamente você só poderá desenvolver com um Mac. Sim, é um computador bem mais caro do que o valor que estamos habituados. Você pode comprar seu Mac em diversos lugares, sendo que alguns apresentam até preços melhores que na própria Apple Store. No caso de você ser estudante, há um desconto significativo de 10%, fique atento:

http://store.apple.com/br/browse/home/education_routing

Atualmente o iOS SDK vem junto com a ferramenta que auxilia a desenvolver o código, a IDE, que se chama Xcode. Você fará apenas um único download que conterá ambos. Veremos esse processo de instalação no próximo capítulo.

1.2 Como o livro está organizado e focado

Este livro aborda os pilares fundamentais de desenvolvimento para iOS, cobrindo desde a estrutura básica de um aplicativo, passando por detalhes da principal ferramenta de desenvolvimento, o Xcode. Também veremos alguns tópicos mais avançados, como requisições de rede e execução concorrente de tarefas, além de detalhes da linguagem. Para você ter um melhor proveito desse livro, é importante já possuir uma boa experiência com uma outra linguagem de programação, e que se sinta confortável para utilizar estruturas de controle e atuar na solução de problemas lógicos.

Uma das principais preocupações que tive ao escrever o livro foi balancear a profundidade com a qual assuntos mais básicos e fundamentais são abordados, porém sem deixar de lado os leitores mais ávidos por detalhes técnicos. A apresentação de funcionalidades e recursos, inclusive da ferramenta de desenvolvimento, muitas vezes será feita de maneira orgânica, contextualizada em diversos aplicativos, o que possibilita a criação de exemplos relevantes e práticos. Foi dada muita atenção para a criação de material que lhe instigue a querer sempre continuar adiante. Todo feedback é mais que bem vindo.

Praticar é o segredo. Faça testes, use sua criatividade. Não se atenha aos diversos exemplos que temos aqui. Apesar de serem muitos, aproveite para criar e ir além.

1.3 Códigos para download e lista de discussão

Todos os projetos apresentados no livro estão disponíveis para download no endereço https://github.com/rafaelsteil/livro-ios-exemplos, incluindo as eventuais correções e modificações. Além disso, criamos uma lista para perguntas e respostas tanto sobre o livro quanto sobre programação para iOS em geral. O acesso é livre, através do seguinte endereço:

https://groups.google.com/d/forum/programacaoios

Capítulo 2

Hello World, seu primeiro programa em iOS

Aplicativos para iOS seguem uma estrutura relativamente simples, se formos comparar com aplicações que rodam em um browser ou no desktop. Além disso, ao contrário de outros sistemas mobile, como Android, detalhes no iOS são mais padronizados, existindo uma quantidade muito menor de dispositivos e resoluções de tela, o que simplifica muito a vida na hora de criar aplicativos. Quem já fez aplicações para a web sabe o quão difícil é criar um sistema que funcione bem em diversos tamanhos de telas, configurações variadas dos computadores e, além disso tudo, em diferentes browsers. Com iOS estes problemas são muito menores, pois se você fizer um aplicativo para iPhone, poderá ter a certeza que ele rodará da mesma maneira para todos os usuários. Claro, a medida em que o sistema vai evoluindo e novas versões são lançadas no mercado, temos que lidar as variações de processador, memória e densidade de tela.

Compreendendo mais a fundo resoluções e densidades de pixel

Uma leitura muito interessante é o post "Pixels, pixels ou pixels?", na URL http://bit.ly/A6odLG.

O código-fonte deste capítulo está disponível na pasta "HelloWorld" (lembrando que o endereço do site com os códigos está na introdução do livro).

2.1 Instalando a ferramenta de desenvolvimento - Xcode

A Apple disponibiliza gratuitamente todas as ferramentas e documentação para desenvolver para iOS, incluindo um simulador de iPhone e iPad que funciona incrivelmente bem, além do Xcode, que é onde todo o código é escrito e compilado. A maneira mais fácil de obter o Xcode é através da Mac App Store, disponível desde versão 10.6.6 do OS X.

Basta procurar por "Xcode" e instalar gratuitamente. Ou acesse o website da ferramenta e clique em "*View in Mac App Store*": https://developer.apple.com/xcode/

Vale lembrar que o Xcode já contém todas as ferramentas do SDK para o iOS (e para o OS X também!).

2.2 Seu primeiro programa

Nesta seção vamos ver como criar uma simples aplicação para iPhone, do início ao fim. O principal intuito é introduzir as principais telas e funcionalidades do Xcode, e ir se familiarizando com o editor de códigos e sintaxe da linguagem. Para todos os projetos deste livro foi utilizada a versão 5 do Xcode.

Crie um novo projeto no Xcode (*File -> New -> Project*), e escolha o template *Single View Application*, conforme a figura 2.1, e depois clique em *Next*:

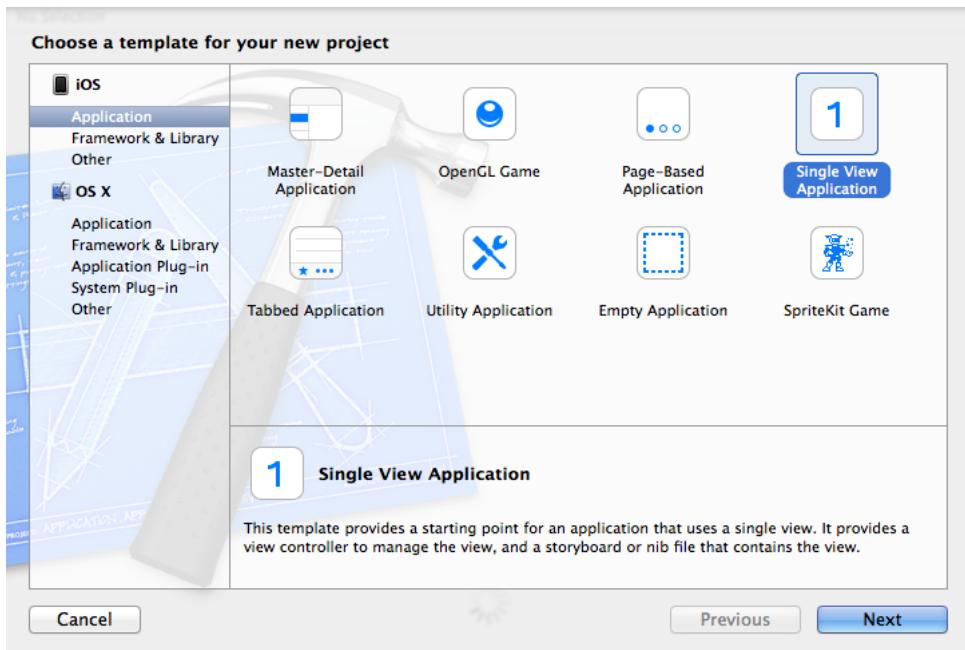



Figura 2.1: Selecionando o tipo de projeto a ser criado

Na próxima tela, demonstrada na figura 2.2, devemos inserir algumas informações básicas a respeito do projeto, como nome (*Product Name*), identificador da empresa (*Company Identifier*), e um prefixo para a classe (*Class prefix*). Insira os valores "*HelloWorld*", "*com.teste*" e "*HW*" respectivamente.

O *Product name* é o nome do projeto (que depois pode ser customizado), o campo *Company Identifier* é o identificador do projeto que será utilizado na identificação técnica do produto quando você for cadastrá-lo junto à Apple, no iTunes Connect. Todo o processo de registro do aplicativo e utilização do iTunes Connect serão abordados mais adiante no livro. Não há uma regra específica para o valor deste campo, exceto que ele deve ser único. Porém, como convenção, costuma-se utilizar algo parecido como os packages do Java, normalmente o endereço web da empresa, seguido do nome do projeto. Repare que existe um label embaixo chamado *Bundle Identifier*, que é a junção do *Company Identifier* com o *Product Name*. Estas informações serão úteis mais adiante quando formos cadastrar o produto no iTunes Connect, mas por enquanto não se preocupe muito com os valores que for utilizar, já que estamos apenas criando um aplicativo de testes.

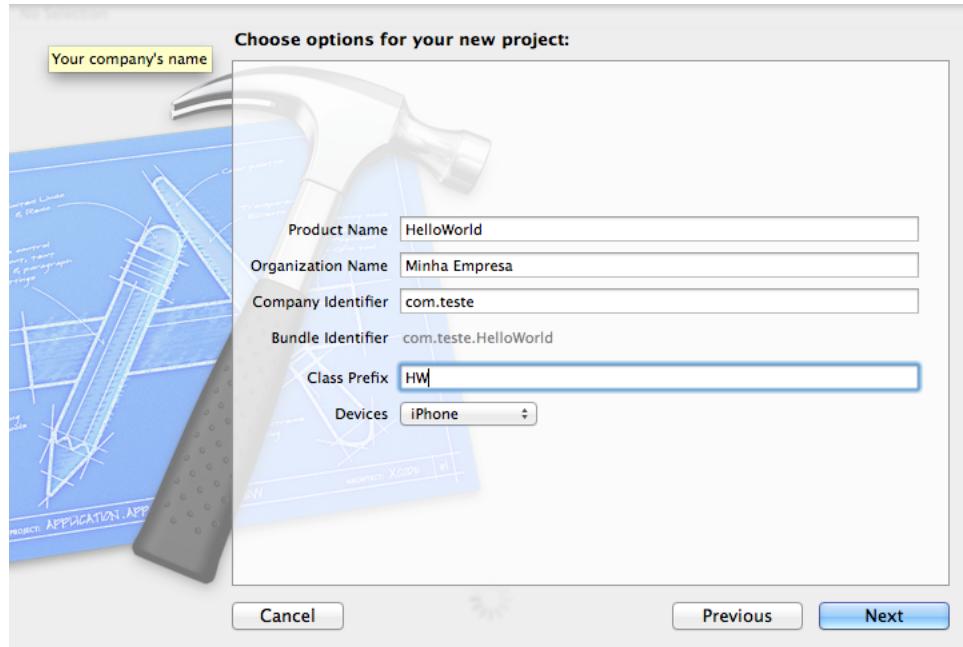



Figura 2.2: Informações sobre o projeto

Por sua vez, o campo opcional *Class Prefix* denota uma boa prática de programação Objective-C, que é a de colocar um prefixo no nome de todas as classes para evitar conflitos de nomes. Quem está acostumado com linguagens como Java, C# ou mesmo C++, tem um aliado a mais para evitar este problema, que são os packages ou namespaces. Em Objective-C não existe este conceito, portanto precisamos garantir que não existam nomes duplicados em nosso projeto e — muito importante — em nenhuma outra biblioteca de que façamos uso. Caso ocorra um conflito de nomes, o compilador irá acusar um erro e terminar o processo de compilação do programa. No caso deste exemplo, foi utilizado o prefixo *HW*, de *Hello World*.

Por último, selecione "*iPhone*" no campo "*Devices*", pois este aplicativo será apenas para iPhone. As outras opções possíveis são "*iPad*" e "*Universal*", que é um tipo de projeto que pode rodar tanto em iPhone quanto em iPad.

Clique em *Next*, e o Xcode pedirá para você salvar o projeto. Escolha o diretório de sua preferência e clique em "*Create*"

Neste ponto já temos o projeto criado com as definições padrão, e você deverá ver uma tela parecida com a da figura 2.3.

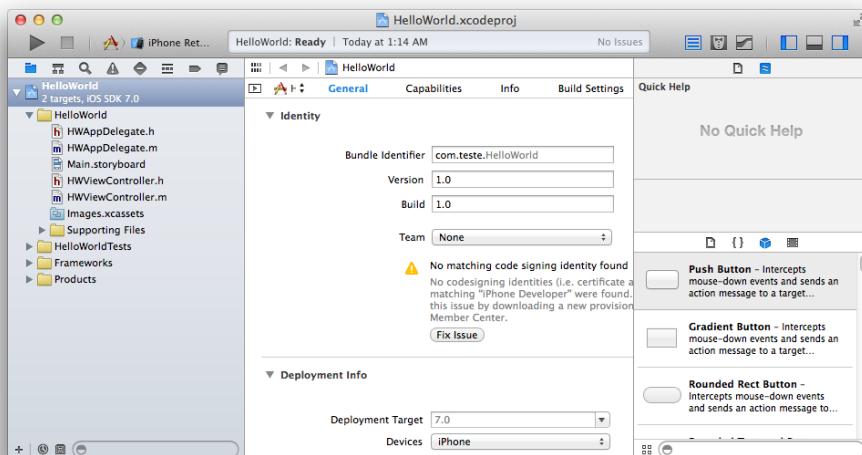



Figura 2.3: Tela do Xcode após a criação do projeto

O projeto foi criado com 5 arquivos:

- `HWAppDelegate.h` e `HWAppDelegate.m`
- `HWViewController.h` e `HWViewController.m`
- `Main.storyboard`

A classe `HWAppDelegate` é a responsável por, dentre outros, eventos de inicialização e término do aplicativo. É esta classe que inicializa o nosso controller principal e permite que o usuário possa utilizar o aplicativo.

A classe `HWViewController` é o nosso controller principal, responsável por gerenciar diversos eventos do aplicativo — como rotação, navegação para outros controllers e gerenciamento da view —, e é uma das classes com que mais trabalharemos ao construir aplicativos para iOS.

Por último, o arquivo `Main.storyboard` é utilizado para desenhar a tela do aplicativo. Mais adiante no livro veremos a fundo o funcionamento de storyboards, por enquanto basta saber que é nele onde as telas serão criadas.

2.3 Dando vida ao aplicativo

A ideia deste aplicativo é esconder e mostrar uma tela (*view*) através do toque em um botão. No fim, teremos visto:

1) Como customizar a tela do controller principal
2) Como adicionar elementos na tela utilizando o Storyboard
3) Como lidar com eventos de botões
4) Como fazer uma animação simples

Editor visual de telas

Até a versão 3 do Xcode, as telas eram representadas apenas por arquivos com a extensão `.xib`, e editados através de um programa à parte chamado Interface Builder. Porém, a partir da versão 4, ele foi integrado diretamente com o Xcode. Existem diversas formas de integrar a parte visual de um arquivo `.xib` com o código-fonte propriamente dito, o que pode causar um pouco de confusão no início do aprendizado.

Além disso, a partir do Xcode 4.2 a Apple introduziu o conceito de Storyboards, possibilitando assim a criação e gerenciamento de todas as telas do aplicativo em um único arquivo, ao invés de diversos "xibs". De qualquer maneira, Xibs continuam sendo uma maneira totalmente válida de criar interfaces.

Aqui no livro detalharemos os passos necessários para que você não fique perdido com a quantidade de informações, porém não se apegue a uma única maneira de desenhar as interfaces, e não desanime caso pareça difícil no início. Rapidamente você verá que o fluxo é bastante lógico, e passará a trabalhar com mais produtividade.

Para começar, vamos criar os elementos gráficos que serão manipulados. Para isso, clique no arquivo `Main.storyboard`, e você deverá ver uma tela como a da figura 2.4.

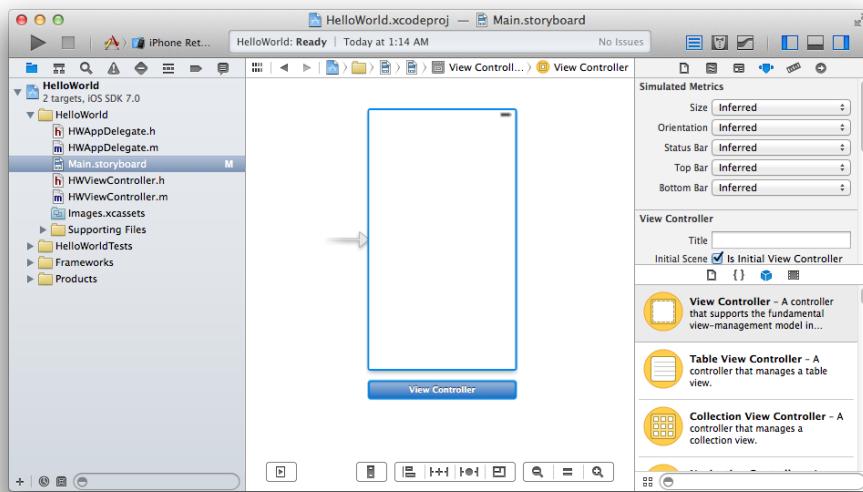



Figura 2.4: Tela inicial do Interface Builder

Os componentes que podem ser adicionais na view, como outras views, botões, seletores de opções, campos de texto e afins estão disponíveis na **Object Library**, acessível através do menu `View -> Utilities -> Show Object Library`, que por padrão estará localizada no canto inferior direito, conforme a imagem 2.5.

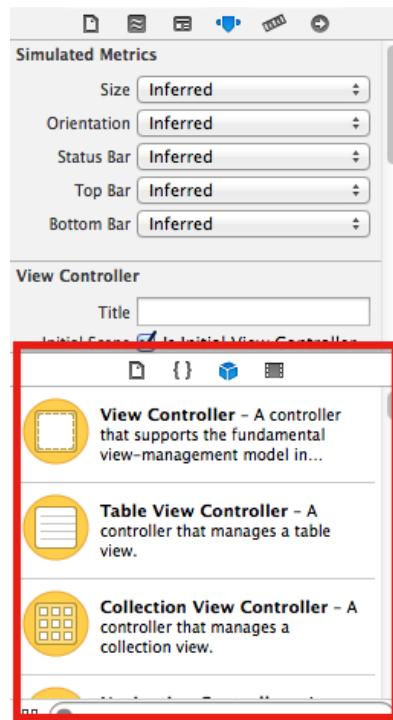



Figura 2.5: A Object Library

Nesta lista, selecione o componente `View` (que é uma instância da classe `UIView`) e arraste-o para a tela do iPhone. Ao fazer isso você notará que o Xcode tentará adicionar a view de tal forma que preencha todo o espaço disponível, porém é possível mudar o tamanho sem problemas. Feito isso, selecione o *Size Inspector* através do menu *View -> Utilities -> Show Size Inspector*, que deverá estar localizado no canto superior da tela. Esta janela permite alterar diversas propriedades do componente, como tamanho e posição, cor de fundo e transparência. Por hora, mude o tamanho nos campos `Width` e `Height` para `280` e `150`, respectivamente, e posicione-o no `X` e `Y 20`. Você pode fazer esta operação com o mouse também.

Repare que tanto o fundo da tela quanto a view que acabamos de adicionar tem fundo branco - um fantasma numa tempestade de neve. Para trocar a cor de fundo da view selecione o *Attributes Inspector* em *View -> Utilities -> Show Attributes Inspector*, e em seguida mude o campo “*Background*” para preto, ou qualquer outra cor que lhe agradar.

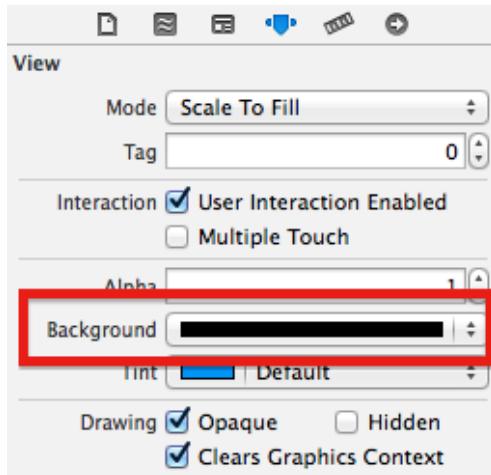



Figura 2.6: Localização da cor de fundo do componente, no Attributes Inspector

O próximo passo é adicionar dois botões, um para esconder a view, e outro para mostrá-la novamente. Os botões são do tipo `UIButton`. Na paleta de componentes, eles estão listados como *Button*. Botões podem ser transparentes também, assim como utilizar uma imagem como fundo. O primeiro botão será chamado *Esconder*, e o segundo será chamado *Mostrar*. Existem duas formas de fazer isso pelo Interface Builder: dando dois cliques em cima do botão e inserindo o texto; ou acessando *View -> Utilities -> Show Attributes Inspector* e preenchendo a propriedade `Title`.

Attributes Inspector

O *Attributes Inspector* permite definir diversas propriedades dos componentes, variando de acordo com o tipo de cada um. Por hora, apenas o título do botão nos interessa.

Após ter feito estes passos, a tela deverá estar como a figura 2.7.

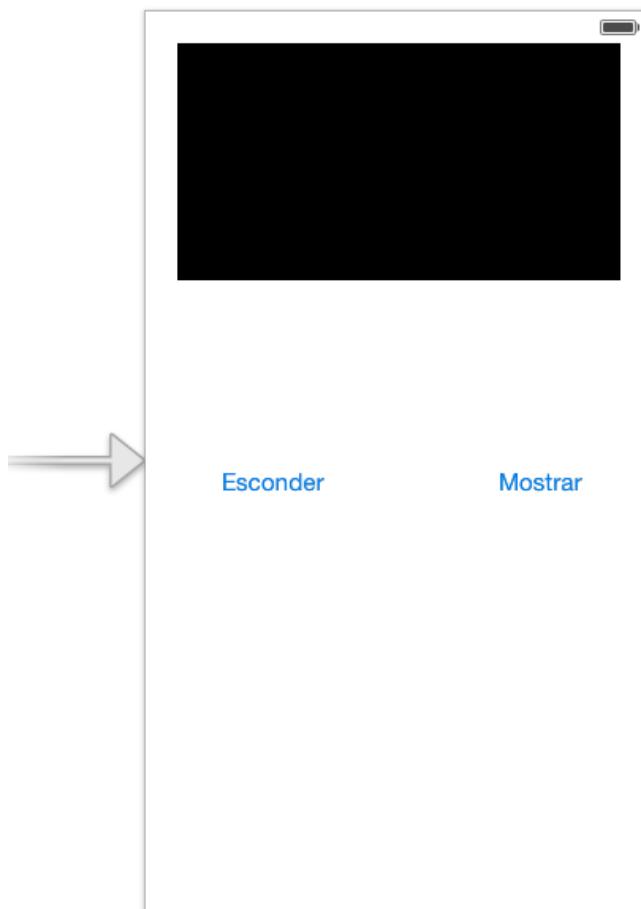



Figura 2.7: Estado da view após adicionar os componentes

2.4 Associe sua tela com o código

Uma vez tendo criado a tela, é necessário associar os componentes com o código propriamente dito. Para isso o Xcode fornece algumas facilidades de arrastar e soltar que nos livram de escrever código repetitivo o tempo todo. Os passos são os seguintes:

1) Ativar o *Assistant Editor* do Xcode

2) Esconder telas desnecessárias

3) Ligar os componentes com o código-fonte

O primeiro passo é ativar o *Assistant Editor* ("Editor assistente", em uma tradução literal) do Xcode, que é um modo de trabalho no qual podemos ver ao mesmo tempo tanto a interface gráfica quando o código-fonte. Acesse o menu *View -> Assistant Editor -> Show Assistant Editor* (tecla de atalho `Option + Command + Enter`), o que deverá mostrar uma nova tela com o arquivo `HWViewController.h` ao lado da interface gráfica que acabamos de montar.

> **Teclas de atalho**
>
> Repare que muitos dos menus têm teclas de atalho, que são grandes aliadas dos programadores. Mais adiante no livro veremos uma relação de diversas teclas de atalho bastante úteis no dia a dia.

A figura 2.8 mostra como fazer o mesmo procedimento utilizando botões existentes no Xcode.

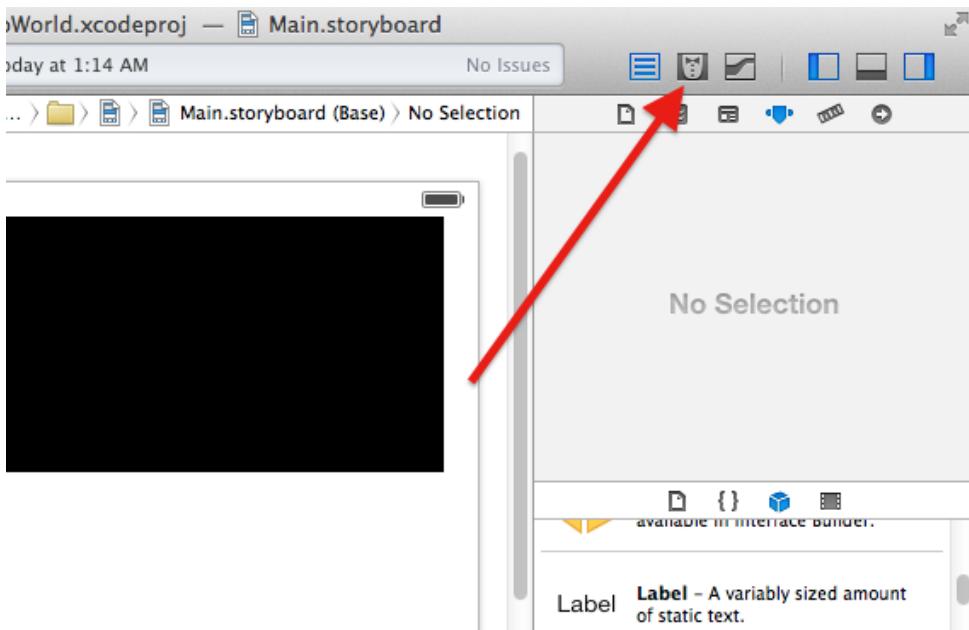



Figura 2.8: Botão para ativar o editor assistente

Agora, se você estiver com as configurações padrões do Xcode, a tela deverá estar uma confusão como a da figura 2.9, especialmente se o seu monitor não for muito grande.

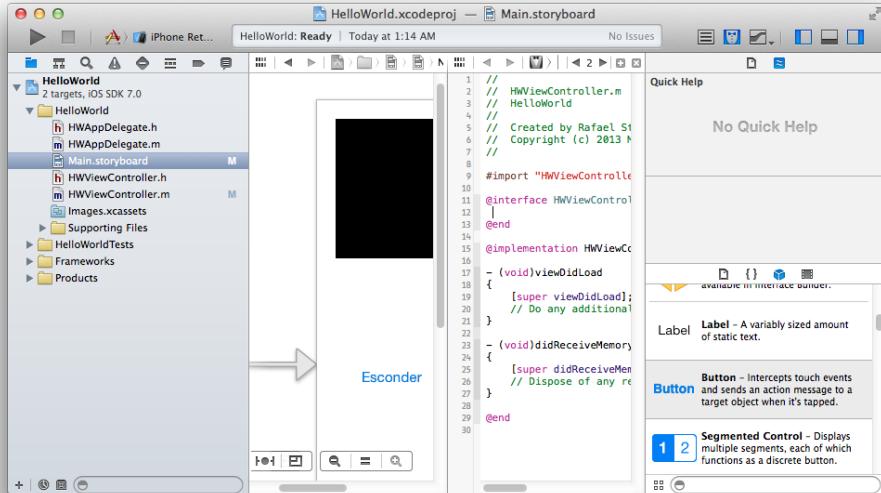
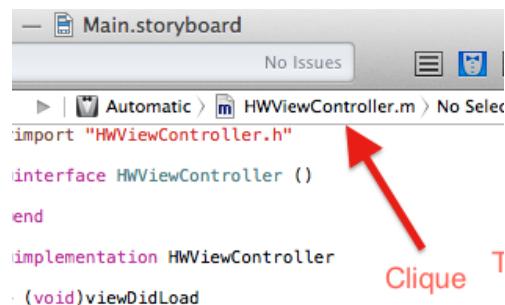
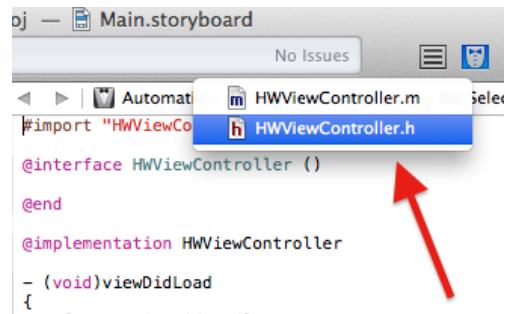



Figura 2.9: Editor bastante confuso do Xcode

Podemos simplificar isso escondendo as telas *Navigator* e *Utilities*, que ficam à esquerda e à direita, respectivamente. Para isso, selecione o menu *View -> Navigators -> Hide Navigator* e *View -> Utilities -> Hide Utilities*.

Verifique se o arquivo fonte selecionado é o .h

Importante: os passos a seguir devem ser feitos com o arquivo " `HWViewController.h`" selecionado no *Assistant Editor*, porém algumas vezes o Xcode abre inicialmente o arquivo " `.m`". Para efetuar a troca, clique no nome do arquivo na barra superior de navegação, e selecione " `HWViewController.h`", conforme exemplificado nas imagens 2.10 e 2.11

Figura 2.10: Barra de seleção de arquivo

Figura 2.11: Selecione o arquivo .h

Para conectar cada botão com uma ação no código-fonte faça o seguinte:

- Selecione o botão *Esconder* no Interface Builder
- Segure a tecla `CTRL`, depois clique com o botão esquerdo do mouse e arraste. Você verá que uma linha aparece na tela.
- Arraste a linha para o código-fonte à esquerda (arquivo `HWViewController.h`), posicionando-a logo abaixo da declaração `@interface`, e solte o mouse. Veja a figura 2.12 para referência visual destes passos.
- Deverá aparecer uma janela flutuante como a da figura 2.13. Selecione a opção *Action* em *Connection*, e no campo *Name* insira o valor `hideView`, que é o nome do método que criaremos para manipular o aplicativo.
- Deixe o resto dos campos com os valores padrão, e clique no botão *Connect*
- Repita a operação para o botão *Mostrar*, e no campo *Name* preencha com `showView`

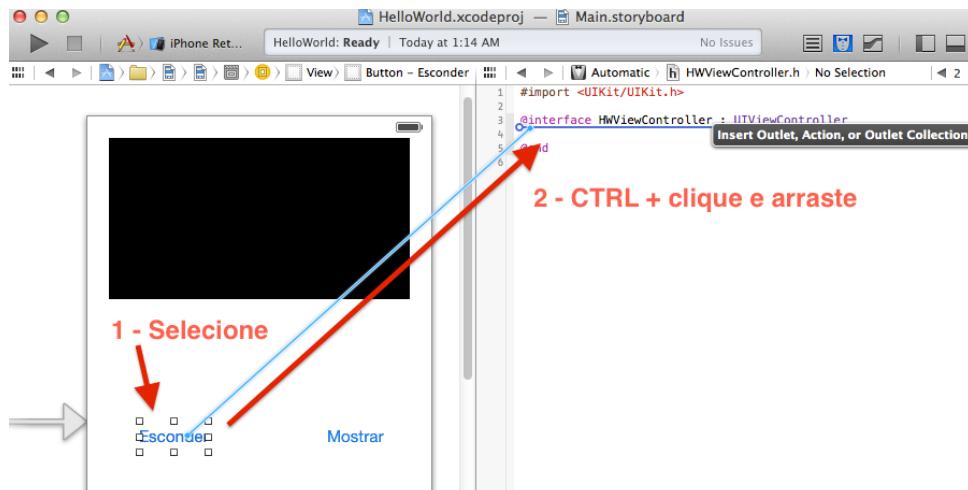



Figura 2.12: Ligando a ação do botão ao código

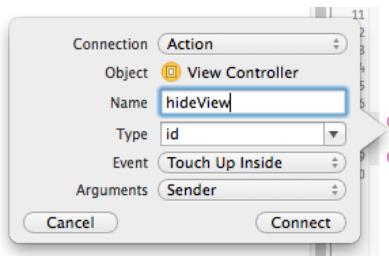



Figura 2.13: Popup para inserir ação a um botão

A última coisa que falta para que possamos começar a colocar a mão no código é associar a `UIView` com o código. Faça o mesmo procedimento realizado para os botões (selecionar, depois `CTRL` + arrastar), porém selecione *Outlet* no campo *Connection*, e em *Name* insira o valor `workingView`. O resultado final do arquivo `HWViewController.h` deverá ser como o da listagem abaixo:

```
#import <UIKit/UIKit.h>

@interface HWViewController : UIViewController
- (IBAction)hideView:(id)sender;
- (IBAction)showView:(id)sender;
@property (weak, nonatomic) IBOutlet UIView *workingView;

@end
```

Outlets

Outlet é a denominação que o Interface Builder utiliza para se referenciar a um componente no arquivo `.storyboard` ou `.xib` que é referenciado no arquivo `.h`, onde é representado pela palavra-chave `IBOutlet`. Ele é tratado de forma especial pelo Xcode. Sem os outlets, não seria possível acessar os componentes criados.

Estamos na reta final do nosso primeiro aplicativo para iOS. O que devemos fazer agora é criar o código que irá esconder e mostrar a view de acordo com os botões. Para isso precisamos trabalhar no arquivo `HWViewController.m`, que é onde fica o código de fato.

Arquivos .h e .m

Ao contrário de linguagens como Java e C#, o Objective-C (assim como C e C++) requer um arquivo para as definições gerais da classe, e outro para o código de fato, tendo a extensão `.h` e `.m` respectivamente. No capítulo sobre a linguagem veremos mais a fundo o funcionamento deles.

Para alternar para o arquivo `.m` (`HWViewController.m`), abra novamente o painel Navigator (*View -> Navigators -> Show Navigator*, ou `Command+0`) e selecione o arquivo na listagem. Caso queira fechar o *Assitant Editor* e voltar para o modo de edição com um único arquivo por vez, selecione *View -> Standard Editor -> Show Standard Editor* (ou `Command+ENTER`).

Repare que o código do arquivo `HWViewController.m` já contém o esqueleto dos métodos `hideView` e `showView`, que foram criados pelo Xcode quando conectamos os componentes do arquivo `Main.storyboard` com o arquivo `.h`.

2.5 Escondendo e mostrando a view

Como a nossa `workingView` já se inicia visível, vamos primeiro fazer o código para escondê-la. Para isso, implemente o método `hideView` conforme exemplo abaixo:

```
- (IBAction)hideView:(id)sender {
    self.workingView.alpha = 0;
}
```

É um código bastante simples e autoexplicativo, em que mudamos a opacidade do componente para 0, tornando-o assim invisível ao usuário. Já o código para mostrar novamente o componente consiste em fazer a operação contrária — ou seja, mudar a propriedade `alpha` para o valor *1*, conforme mostra a listagem abaixo:

```
- (IBAction)showView:(id)sender {
    self.workingView.alpha = 1;
}
```

Para compilar e rodar o aplicativo no simulador do iPhone, selecione o menu `Product -> Run` (`Command+R`). O resultado deverá ser como o da figura 2.14.

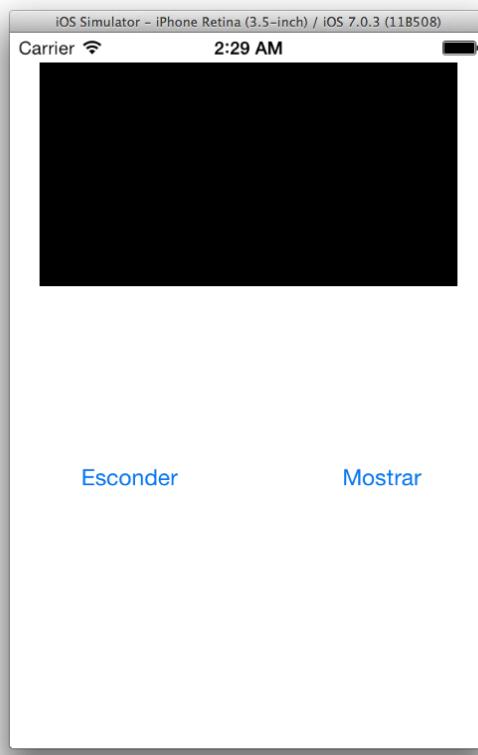



Figura 2.14: Resultado do primeiro aplicativo para iPhone

2.6 Adicionar animação

Para finalizar este primeiro aplicativo, podemos tornar as operações de esconder e mostrar o componente algo visualmente mais agradável, através de animações. Para tanto, modifique os métodos conforme o exemplo abaixo:

```
- (IBAction)hideView:(id)sender {
    [UIView beginAnimations:nil context:nil];
    self.workingView.alpha = 0;
    [UIView commitAnimations];
}
```

```
- (IBAction)showView:(id)sender {
    [UIView beginAnimations:nil context:nil];
    self.workingView.alpha = 1;
    [UIView commitAnimations];
}
```

Rode no simulador novamente (`Command+R`) e veja a diferença. Muito melhor, não?!

2.7 Bônus - Acessar a documentação de dentro do Xcode

A API do iOS é enorme, e simplesmente não há como sabermos de memória o nome de todas classes, métodos, propriedades e as infinitas regras que os regem. Para isso existe a documentação, e o Xcode torna bastante fácil o acesso a ela: basta abrir o painel *Quick Help* ("Ajuda rápida", literalmente) através do atalho `Option + Command + 2`, ou através do menu *View -> Utilities -> Show Quick Help Inspector*, e depois clicar no editor de códigos em cima da classe a cuja documentação se deseja ter acesso, conforme mostra a figura 2.15.

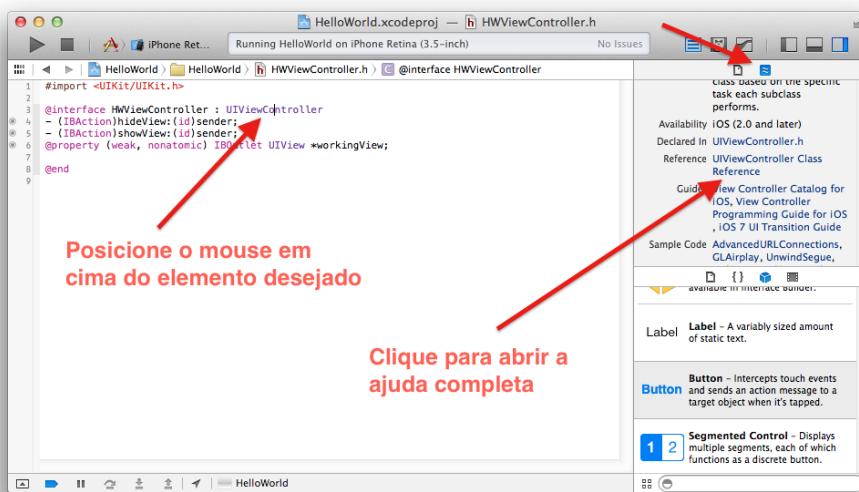



Figura 2.15: Acessando a ajuda rápida

Capítulo 3

Mais Objective-C e Xcode

Agora que já vimos a estrutura básica de um aplicativo iOS, tendo passado pelo processo de criação de um projeto, algumas teclas de atalho e funcionalidades do Xcode, vamos dar um passo adiante e criar um aplicativo um pouco mais elaborado. Ao mesmo tempo, exploraremos mais a fundo o Xcode e, principalmente, a linguagem Objective-C. Detalharemos mais a linguagem de programação, porém sem entrar em aspectos avançados, o que é feito no capítulo 11.

O aplicativo que iremos criar consiste no cadastro de empresas para um catálogo, mostrando a listagem de todas empresas cadastradas cada vez que um novo registro é inserido. A figura 3.1 mostra onde chegaremos com o nosso aplicativo. Durante a construção deste, não deixe de testar ideias e deixar sua curiosidade ajudá-lo a aprender mais.

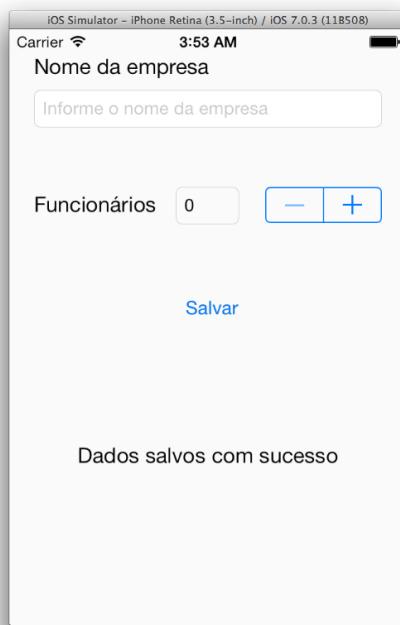



Figura 3.1: O aplicativo que iremos construir

O código-fonte deste capítulo está disponível na pasta "CatalogoEmpresas" (lembrando que o endereço do site com os códigos está na introdução do livro).

3.1 Criando o projeto

Abra o Xcode e crie um novo projeto através do menu *File -> New -> Project* (tecla de atalho *SHIFT + Command + N*) e selecione o tipo *Single View Application* e clique no botão *Next*. Na próxima tela insira o nome `CatalogoEmpresas` no campo *Product Name*, em *Company Identifier* coloque `com.teste`, deixe o campo *Class Prefix* em branco e selecione "*iPhone*" em *Device Family*. Veja a figura 3.2 para referência. Clique em *Next* para salvar o projeto.

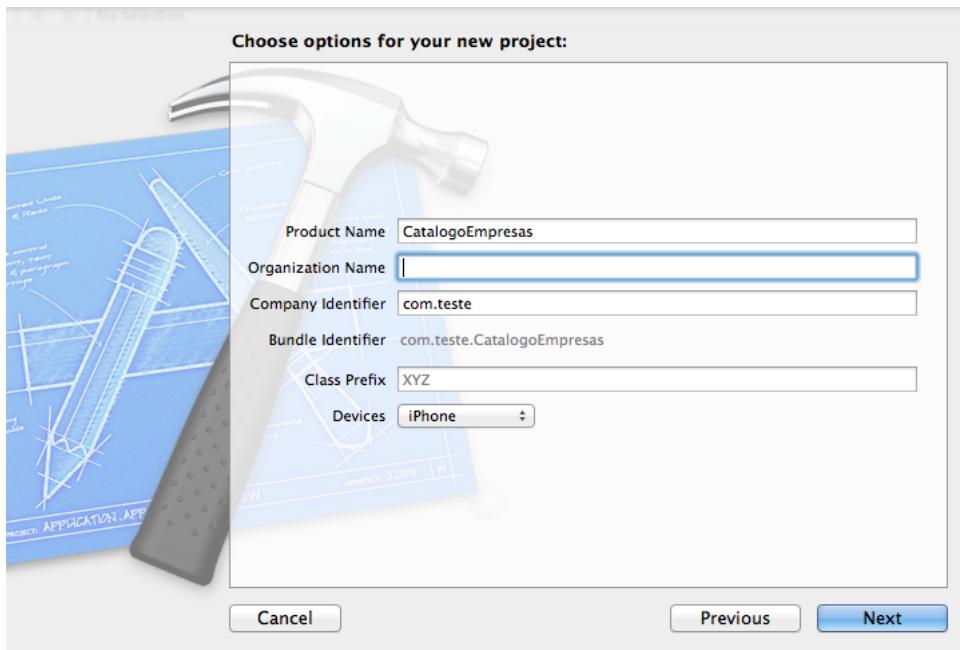



Figura 3.2: Opções do novo projeto

Os diferentes templates de projetos

Apesar do nome, o template de projeto *Single View Application* (ou "aplicação com uma única view") não é limitado a uma única tela, mas sim que o código inicial que o Xcode irá criar para você terá uma tela inicial, ficando a cargo do desenvolvedor criar o que mais precisar. Os outros tipos de templates seguem a mesma lógica, porém adicionando por padrão algumas coisas a mais, como o a *Tabbed Application*, que cria a estrutura base para um projeto que use uma "tab bar" (aquele componente com uma barra inferior com vários ícones), ou a *Utility Application*, que cria um projeto já com duas views e um botão para alternar entre elas.

Nos exemplos deste livro iremos sempre utilizar a *Single View Application* por ser a mais prática para os propósitos apresentados. Aliás, este muito provavelmente será o template que você mais utilizará quando criar seus próprios aplicativos.

3.2 Entenda melhor a interface do Xcode

A figura 3.3 mostra a estrutura geral do Xcode, com os possíveis painéis de trabalho abertos, sendo que cada um destes painéis é utilizado para diferentes propósitos — nessa figura, apresentamos apenas uma das combinações possíveis, que é a seguinte:

1) Navegador do projeto, ou apenas *Navigator*, o qual contém a relação de arquivos do projeto (`Command + 1`), erros de compilação, logs, busca e outras funcionalidades. Tecla de atalho: `Command + 1` para abrir, e `Command + 0` para esconder (menu *View* -> *Navigators* para a relação completa)

2) Editor principal de código e criação de telas, chamado oficialmente de *Standard Editor*. Dependendo da configuração pode mostrar mais de um arquivo ao mesmo tempo. Tecla de atalho: `Command + ENTER`. Para alternar para o modo de trabalho em par, no qual a tela fica dividida entre dos arquivos diretamente relacionados, a tecla de atalho é `Command + Option + ENTER`.

3) Painel de debugging ("depuração") e mensagens de log geradas pelo aplicativo. Teclas de atalho: `Command + SHIFT + Y` e `Command + SHIFT + C` (menu *View* -> *Debug Area*)

4) Painel *Utilities*, o qual contém utilidades gerais de acordo com o arquivo aberto no *Standard Editor*, tais como documentação (quando editando código-fonte) ou propriedades de algum componente visual (quando editando arquivos `.xib`)

5) Botões de acesso rápido para os editores e painéis já descritos.

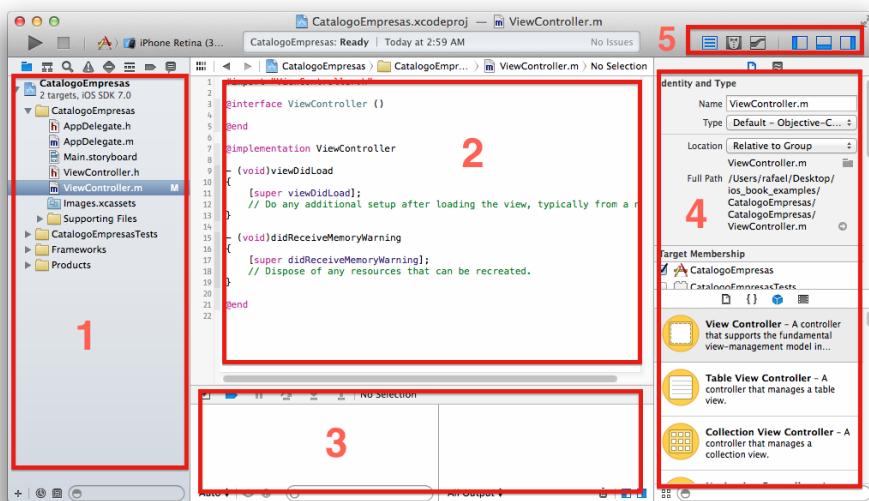



Figura 3.3: Os painéis do Xcode

> **Termos em inglês**
>
> Ao referenciar menus e opções do Xcode utilizaremos primeiro o termo original em inglês e, quando pertinente, uma tradução literal em português, pois desta forma será mais fácil para você procurar ajuda e documentação na Internet, mesmo que não domine totalmente o inglês.

3.3 Criar a tela de inserção de empresa

O primeiro passo é montar a tela para que possamos posteriormente acessar os componentes gráficos no código-fonte. Selecione o arquivo `Main.storyboard` no *Project Navigator* (`Command + 1`) e você será apresentado com a tela que corresponde àquela que será vista pelo usuário ao rodar o aplicativo. O objetivo aqui é construí-la da mesma forma como foi apresentado na figura 3.1 no início do capítulo.

Passos detalhados

Ao escrever este livro nos preocupamos bastante para manter um bom balanceamento entre mostrar passo a passo como realizar determinadas coisas, versus apresentar uma instrução mais curta e direta. Nos capítulos iniciais, e especialmente neste em que apresentamos em mais detalhes o funcionamento do Xcode e Objective-C, é feito bastante uso de imagens para guiar o leitor para determinados menus e botões (além de explicar os atalhos de teclado), pois o Xcode utiliza os mesmos painéis para diferentes tarefas, o que às vezes pode ficar bastante confuso.

Vamos modificar a cor de fundo da tela principal, colocando um tom de cinza claro. Clique em qualquer parte da tela do aplicativo e abra o *Attributes Inspector* ("Inspetor de atributos") utilizando a tecla de atalho `Option + Command + 4`, ou através do menu *View -> Utilities -> Show Attributes Inspector*. Veja a figura 3.4 para referência.

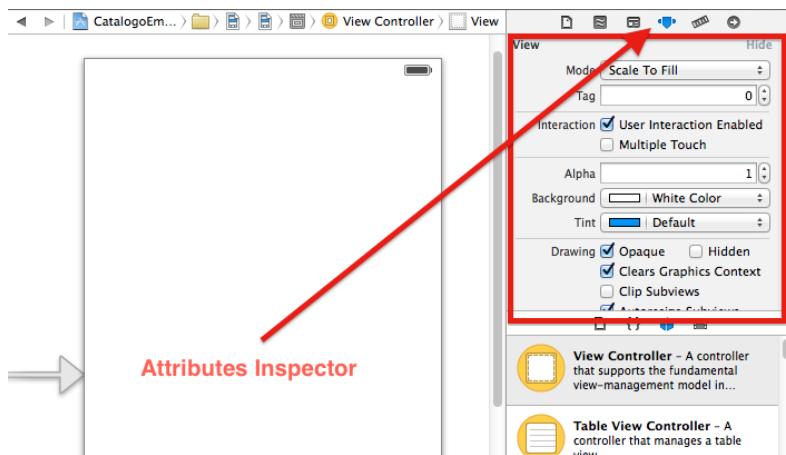



Figura 3.4: Abrir o Attributes Inspector

Para modificar a cor de fundo clique no item "*Background*" para abrir a janela flutuante "*Colors*", e selecione a opção "*Color Sliders*" na barra superior. Em seguida, selecione "*RGB Sliders*" e insira o valor "250" para os três campos, conforme mostra a figura 3.5. Feche a janela "*Colors*".

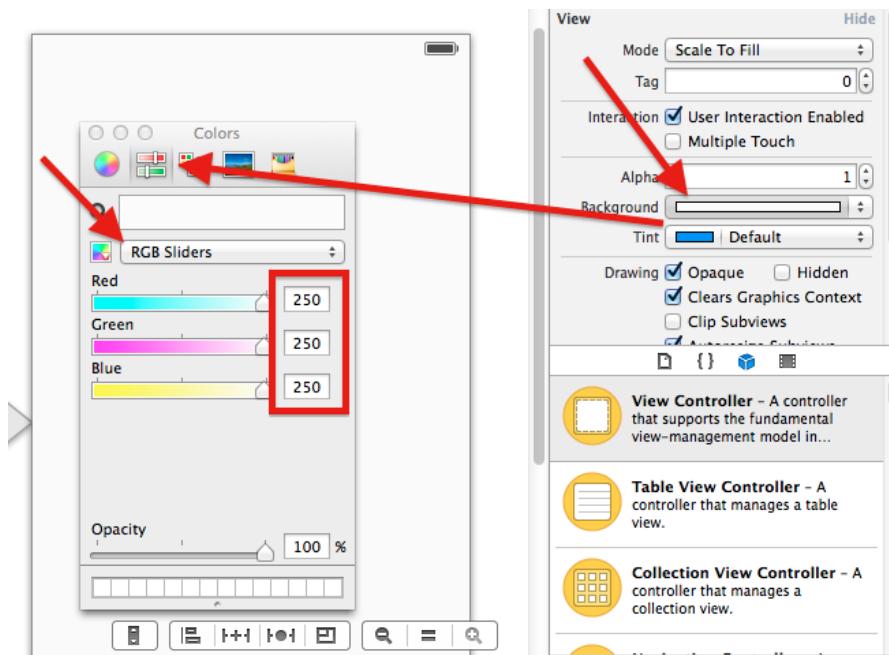



Figura 3.5: Modificando a cor de um componente - no caso, a cor de fundo da tela

3.4 Adicionar componentes visuais

Abra a *Object Library* na parte inferior do painel *Utilities* utilizando a tecla de atalho `CTRL + Option + Command + 3`, ou através do menu *View -> Utilities -> Show Object library*, e adicione os seguintes componentes, posicionando-os para que fiquem como a figura 3.1, mostrada no início deste capítulo.

- *Label* com o texto "`Nome da empresa`" (dê duplo clique no label, ou então preencha a propriedade *Text* no *Attributes Inspector*).
- Um componente *Text Field* para o usuário digitar o nome da empresa. No *Attributes Inspector* localize a opção *Placeholder* e digite "`Informe o nome da empresa`" - um *placeholder* é um valor padrão que o *Text Field* irá mostrar enquanto o usuário não inserir algum texto.
- *Label* com o texto "`Funcionários`".

- Outro componente *Text Field*, para a quantidade de funcionários. No *Attributes Inspector* localize a seção *Control* e desmarque o checkbox *Enabled*, pois queremos que o usuário modifique o campo utilizando o componente *Stepper* (veja abaixo). No campo *Text* insira o valor "0", que será a quantidade inicial de funcionários da empresa.

- Um componente *Stepper*: que será utilizado para aumentar ou diminuir a quantidade de funcionários.

- Um botão *Button* com o texto " `Salvar`" (propriedade *Title*, ou duplo clique no componente).

- Um componente *Label* com o texto "Dados salvos com sucesso", alinhado ao centro (propriedade *Alignment* no *Attributes Inspector*), e da largura da tela. Note que ao modificar o tamanho do componente o Interface Builder mostra linhas guia para auxiliar o posicionamento, conforme a figura 3.6.

São muitos componentes! Pratique os diversos tipos, em especiais os que estamos utilizando agora. Eles serão necessário na grande maioria das suas aplicações.

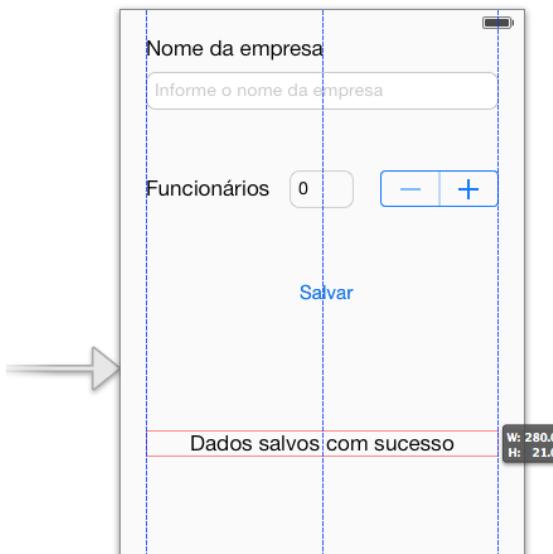



Figura 3.6: Guias do Xcode auxiliando no posicionamento e alinhamento dos componentes

Rode o aplicativo no simulador através da tecla de atalho `Command + R`, ou pelo menu *Product* -> *Run*, para ver o resultado até o momento. Se você interagir com os elementos verá que não acontecerá muita coisa, pois ainda não conectamos os eventos ao código, o que será feito em seguida.

3.5 Atenção para as propriedades simuladas

O *Attributes Inspector* tem uma seção chamada *Simulated Metrics*, que serve unicamente para *simular* alguns elementos de interface, para facilitar o posicionamento dos componentes em certos tipos de aplicativos. Contudo, tenha em mente que, embora o uso das propriedades da seção *Simulated Metrics* resulte em alterações visuais, elas são apenas temporárias, não tendo qualquer impacto prático no aplicativo. Veja a figura 3.7 para referência.

Obs: caso você esteja no arquivo `Main.storyboard`, o box das propriedades simuladas somente irão aparecer caso selecione a barra preta na parte inferior da tela (aquela que tem três botões, um amarelo, outro laranja e o último verde). Se você apenas clicar na "tela" (a parte que em branco que representa o iPhone), o box não aparecerá.

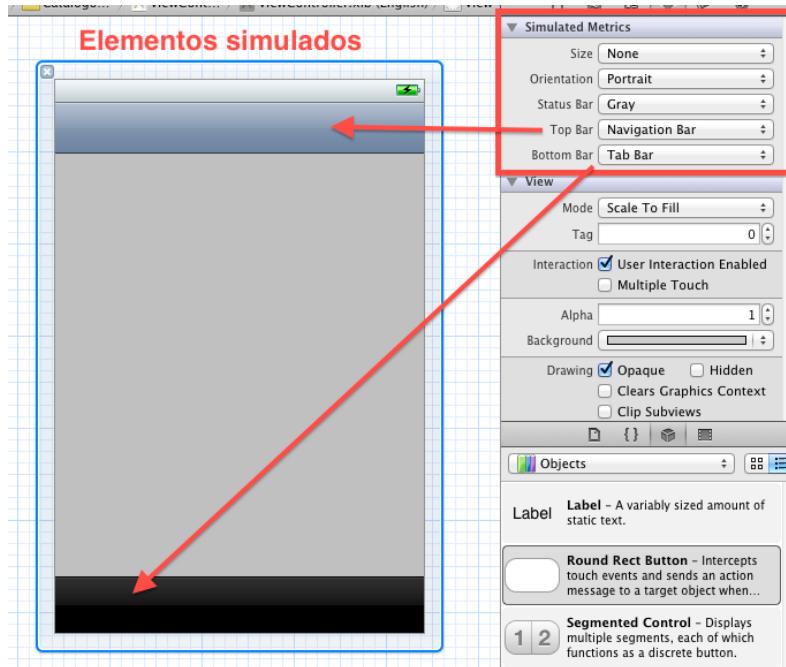



Figura 3.7: Elementos de interface simulados apenas. Eles não irão aparecer no aplicativo de fato.

A possibilidade de *simular* algumas coisas pode ser útil para você construir de maneira mais fácil a tela, se em algum momento (muito provavelmente via código) pretende inserir os componentes "de verdade".

3.6 Conectando os eventos e componentes ao código

Da mesma maneira que fizemos no capítulo anterior, precisamos conectar os componentes gráficos com o código para que seja possível acessá-los de fato — até então, o que é feito no arquivo `Main.storyboard` não tem qualquer relação com código. A primeira coisa a fazer é abrir o *Assistant Editor* (`Option + Command + ENTER`) para visualizar o arquivo `ViewController.h` ao lado do `Main.storyboard`. Lembre-se que, caso o arquivo exibido seja o `ViewController.m` (ao invés do " `.h`"), você deve primeiro alterná-los.

Alternando entre os arquivos .h e .m

É possível alternar rapidamente entre arquivos `.h` e `.m` relacionados utilizando a tecla de atalho `CTRL + Command + Seta para cima` (lembre-se de primeiro clicar na área onde tem algum código), ou então clicando no nome do arquivo na barra de navegação, conforme a figura 3.8.

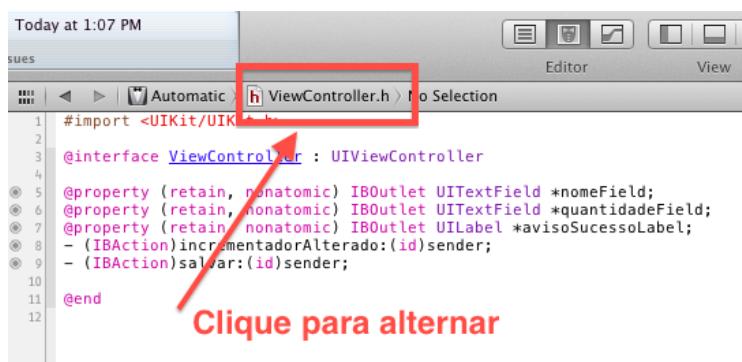



Figura 3.8: Alternando entre arquivos utilizando a barra de navegação

Fechando os painéis desnecessários

É muito fácil deixar diversos painéis abertos no Xcode, aumentando bastante a poluição visual e consequentemente diminuindo a área disponível para trabalhar, especialmente se você está em uma tela pequena. Nesses casos basta esconder o que não for vital, como o *Navigator* (`Command + 0`, ou *View -> Navigators -> Hide Navigator*), o painel de *Debug* (`Command + Shift + Y`, ou *View -> Debug Area -> Hide Debug Area*) e o painel *Utilities* (*`Option + Command + 0`, ou* View -> Utilities -> Hide Utilities::)

A conexão dos componentes deve ser feita da seguinte forma:

- Selecione o campo de texto do nome da empresa, segure CTRL e clique e arraste para o arquivo `ViewController.h`, posicionando entre `@interface` e `end`. Na janela flutuante que abrir selecione o valor "Outlet" na propriedade *Connection* e insira o o valor "nomeField" na propriedade *Name*. Isso irá criar a variável já com as propriedades corretas.

- Repita o mesmo processo para o campo de texto da quantidade de funcionários, colocando como nome o valor "quantidadeField"

- Para o componente *Stepper*, que é aquele com um botão − e outro +, realize o mesmo procedimento de conexão, porém no campo *Connection* selecione o valor "Action" para gerar um método de ação ao invés de criar uma variável simples. No campo *Name* insira o valor "incrementadorAlterado", que será o nome do método que será executado toda vez que o usuário interagir com o componente, e deixe os outros campos com os valores padrão (*Type* "ID", *Event* "Value changed" e *Arguments* "Sender")

- Para o botão salvar, crie a *Connection* do tipo "Action", com o *Name* "salvar", e o aceite o valor padrão do resto dos campos

- Por último, conecte o *label* que tem o texto "Dados salvos com sucesso" criando uma *Connection* do tipo "Outlet", e o *Name* "avisoSucessoLabel"

Neste ponto o arquivo `ViewController.h` deverá estar como o código abaixo:

```
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UITextField *nomeField;
@property (weak, nonatomic) IBOutlet UITextField *quantidadeField;
@property (weak, nonatomic) IBOutlet UILabel *avisoSucessoLabel;
- (IBAction)incrementadorAlterado:(id)sender;
- (IBAction)salvar:(id)sender;

@end
```

Se você abrir o arquivo `ViewController.m` verá que o Xcode já criou para a gente o corpo dos métodos `incrementadorAlterado:` e `salvar:`.

O que exatamente é IBOutlet e IBAction?

Os modificadores `IBOutlet` e `IBAction` são usados unicamente como marcadores para integrar componentes do Interface Builder com o código-fonte em arquivos `.h`. O primeiro é usado quando queremos referenciar o componente no código através de uma variável, e o segundo é usado exclusivamente para ações dos componentes, como o toque em um botão. Por trás dos panos, o que o Xcode faz é analisar o código e automaticamente realizar as devidas conexões entre a interface gráfica e o nosso código.

3.7 Uma classe para representar uma empresa

Agora que já temos a tela criada, é hora de colocar a mão no código, e o primeiro passo é criar a classe `Empresa` para armazenar as informações preenchidas pelo usuário. Para isso, abra a janela de adição de arquivos (`Command + N`, ou então *File -> New -> File...*), e selecione o template *Objective-C class*. Clique no botão *Next*, insira o valor "Empresa" no campo *Class, e em* Subclass of *selecione o item "NSObject". Clique em* Next:: para salvar o arquivo.

Importante

Ao incluir arquivos em qualquer projeto que for trabalhar, certifique-se de que a opção *Targets* esteja selecionada na caixa de diálogo de salvar arquivo conforme mostra a figura 3.9. Isso é necessário para associar o arquivo com o projeto — do contrário, o Xcode ignorá-lo-ia na hora de compilar e empacotar o projeto para distribuição.

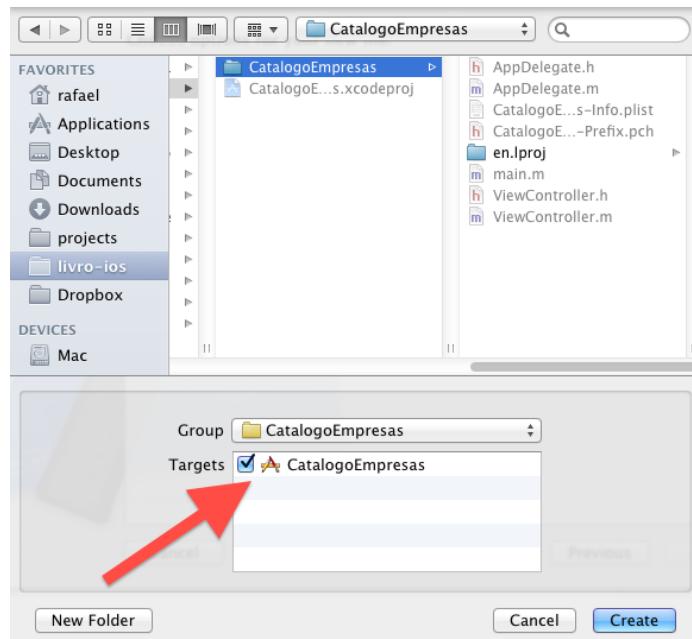



Figura 3.9: Target corretamente selecionado

Alternando novamente entre os painéis

Trabalhar no Xcode é um processo constante entre esconder, mostrar e mudar o tipo dos painéis de acordo com a tarefa a ser feita, de modo que nossa produtividade aumente. No caso desta seção, onde já criamos a tela e vamos focar mais em código, não é necessário manter o arquivo `Main.storyboard` aberto e nem o painel *Utilities*. Alterne para o *Standard Editor* utilizando a tecla de atalho `Command + ENTER`, e esconda o outro painel com o atalho `Option + Command + 0`, e selecione o arquivo `ViewController.m` no *Navigator* (`Command + 1`)

A nossa empresa tem duas propriedades: nome (uma string) e quantidade de funcionários (número inteiro). Abra o arquivo `Empresa.h` e modifique-o para ficar como o código abaixo:

```
1 #import <Foundation/Foundation.h>
2
```

```
3 @interface Empresa : NSObject
4
5 @property (nonatomic, retain) NSString *nome;
6 @property (nonatomic, assign) int quantidadeFuncionarios;
7
8 @end
```

Não é necessário fazer qualquer modificação no arquivo “ `Empresa.m`”.

3.8 Cabeçalhos e implementações

Em Objective-C, a declaração e implementação da classe ficam em arquivos diferentes. A declaração, ou — usando o termo correto — *interface*, fica em um arquivo com a extensão `.h`, de *header* (cabeçalho), enquanto que a implementação fica em um arquivo com a extensão `.m`. Diferentemente de outras linguagens, nas quais se usa apenas um arquivo para tudo o que diz respeito à classe, em Objective-C você sempre precisará da dupla `.h` e `.m`.

No arquivo de cabeçalho (como o `Empresa.h`) fica a declaração da classe, das variáveis de instância, das propriedades, dos métodos de instância e dos métodos de classe.

Nota sobre @interface

A declaração `@interface` é semanticamente muito diferente da mesma palavra-chave no Java e C#, em que representa um determinado contrato que a classe deve seguir. Em Objective-C, `@interface` serve unicamente para declarar a classe, e o equivalente da palavra chave `interface` do Java e do C# é `@protocol`, que veremos em um outro capítulo.

Enquanto o cabeçalho define apenas a estrutura geral da classe, e não o código propriamente dito, o arquivo `.m` contém o “código de verdade”, como arquivo `ViewController.m` ou `Empresa.m`.

Curiosidade sobre a extensão .m

A extensão .m do Objective-C originou-se a partir de “**m**ensagens” (do termo em inglês *messages*), a qual se refere a um dos principais conceitos da linguagem.

3.9 Informando a quantidade de funcionários

Estamos na metade do caminho, mas o resto é só código! Agora que temos a tela e a classe Empresa vamos colocar as ações nos métodos de incrementar a quantidade de funcionários e salvar a empresa em si. Como visto na figura 3.1 a tela contêm dois componentes que devem trabalhar em conjunto para alterar a quantidade de funcionarios: um campo de texto (*UITextField*) e um incrementador (*UIStepper*). O que queremos que aconteça é que quando o usuário tocar em algum dos botões, o valor correspondente apareça no *UITextField* ao lado.

No arquivo ViewController.h criamos uma IBAction chamada incrementadorAlterado:, que é o método a implementar. Abra o arquivo ViewController.m e faça o seguinte:

```
1 - (IBAction)incrementadorAlterado:(id)sender {
2     UIStepper *incrementador = (UIStepper *)sender;
3     self.quantidadeField.text = [NSString stringWithFormat:@"%d",
4         (int)incrementador.value];
5 }
```

O método incrementadorAlterado: segue um padrão bastante comum em Objective-C, que é o de receber um argumento do tipo id chamado sender quando for associado a algum evento. No caso, id pode ser lido como “*qualquer coisa*”, e sender (“remetente”, “quem enviou”) é apenas o nome da variável — você pode modificar para o que achar mais apropriado, apesar de este ser o nome utilizado em toda documentação. Levando isso em consideração, na linha 2 convertemos sender para o tipo UIStepper, pois sabemos que este método será chamado apenas quando o UIStepper da tela for modificado. Já nas linhas 3 e 4 convertemos para string o valor numérico do componente existente na propriedade value, pois o UITextField não aceita números diretamente.

Rode o aplicativo através do atalho Command + R ou através do menu *Product* -> *Run* e clique nos botões - e +, para ver o campo de texto receber os valores.

3.10 Trabalhe com objetos: instanciando uma Empresa

Para criar uma nova empresa precisamos de instâncias da classe `Empresa` e preencher as propriedades `nome` e `quantidadeFuncionarios`. Isso será feito na ação do botão "Salvar", conforme mostrando no código abaixo:

```
1 - (IBAction)salvar:(id)sender {
2     Empresa *e = [[Empresa alloc] init];
3     e.nome = self.nomeField.text;
4     e.quantidadeFuncionarios = [self.quantidadeField.text intValue];
5 
6     NSLog(@"Empresa criada. Nome=%@, funcionários=%d",
7         e.nome, e.quantidadeFuncionarios);
8 }
```

Na linha 2 criamos uma nova instância da classe `Empresa`, enquanto que nas linhas 3 e 4 preenchemos as propriedades com os valores dos componentes da tela. Estas são aquelas mesmas variáveis com `@property` existentes no arquivo `Empresa.h`. Nas linhas 6 e 7 escrevemos os valores no console para fins de debugging.

Compile o projeto com o atalho `Command + B`. Funcionou? A menos que você tenha importado o arquivo `Empresa.h` logo no início do arquivo `ViewController.m`, o Xcode deverá ter acusado erros de compilação informando que a classe `Empresa` não existe. Para resolver este problema, adicione a linha `#import "Empresa.h"` no topo do arquivo `ViewController.m`, recompile o projeto e rode-o com `Command + R`.

O painel de debug e console do Xcode

Rode o aplicativo, insira o nome e quantidade de funcionários e clique no botão salvar. O Xcode deverá mostrar automaticamente uma mensagem de log no painel inferior (caso contrário, abra-o através do atalho `Command + Shift + C`, ou pelo menu *View -> Debug Area -> Activate Console*), conforme a figura 3.10. Como você já associou, a função `NSLog` serve para jogar informações no console do Xcode, o que é útil em diversos momentos durante o ciclo de desenvolvimento e testes do aplicativo.

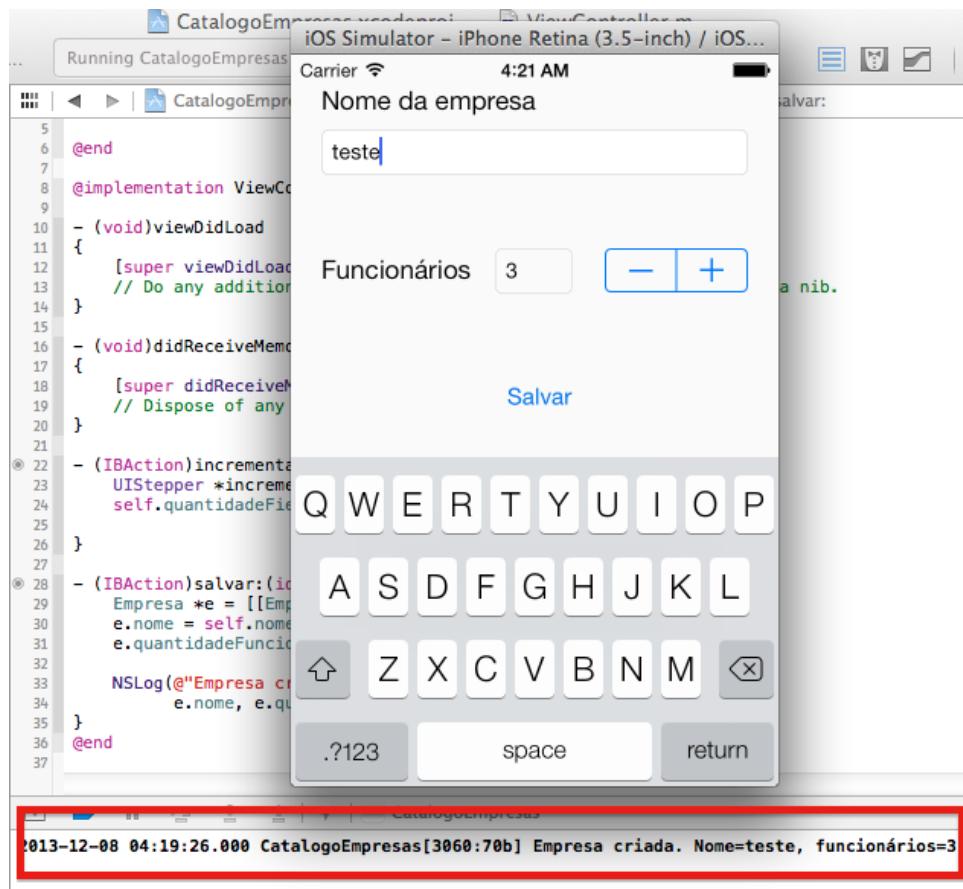



Figura 3.10: Localização do console de debug com saída do NSLog

3.11 Como são as strings em Objective-C?

Strings em Objective-C são representadas pela classe `NSString`, e seus valores devem ser precedidos pelo caractere `@` (arroba). Isso é necessário para se distinguir do array de caracteres da linguagem C (tecnicamente, instruções como `char *`), na qual Objective-C é fundamentada.

A forma correta de criar strings em Objective-C é mostrada abaixo:

```
// OK - utiliza '@'
NSString *nome = @"Rafael Steil";
```

```
// ERRO - o código abaixo não irá funcionar
NSString *ops = "ObjC não funciona assim";
```

3.12 Formatando strings

É possível construir strings utilizando alguns caracteres especiais de formatação através do método `stringWithFormat:`, conforme mostrado abaixo:

```
double faturamento = 1234.567;
NSString *mensagem = [NSString stringWithFormat:
    @"A empresa %@ tem %d funcionários, e faturamento de R$ %f",
    e.nome, e.quantidadeFuncionarios, faturamento];
```

A lista de formatadores é relativamente extensa, porém os mais comuns são:

- `%@` : para qualquer tipo de *objeto*, incluindo outras `NSStrings`
- `%d` : números inteiros (`int` e `unsigned int`)
- `%f` : números de ponto flutuante (`float` e `double`)

A função `NSLog` já aceita por definição esta mesma estrutura de formatação, sem ser necessário a utilização de `stringWithFormat:`.

3.13 Guardando todas empresas em memória

O código que fizemos até agora cria a empresa, mas não a armazena em nenhum lugar, porém nós queremos manter uma relação de todos os registros que foram inseridos e mostrar no console a relação completa a cada nova empresa. Existe uma infinidade de maneiras de realizar esta tarefa, e para o propósito deste capítulo — que é apresentar em maiores detalhes o Xcode e a linguagem Objective-C — vamos nos concentrar em algo simples e prático, como uma lista em memória, utilizando a classe `NSArray`.

NSArray versus array

Apesar de conter a palavra "array" no nome, tecnicamente a classe `NSArray` *não é* um array no conceito formal da linguagem, mas sim se equivale muito mais a uma *lista*, algo como a classe `ArrayList` de Java e C#.

A primeira coisa a fazer é ir para o arquivo `ViewController.h` (tecla de atalho `CTRL + Command + Seta para cima`) e declarar uma variável chamada `catalogo` do tipo `NSMutableArray`, conforme exemplificado abaixo:

```
1  #import <UIKit/UIKit.h>
2
3  @interface ViewController : UIViewController { // Abre chave aqui
4      NSMutableArray *catalogo;
5  } // Fecha chave aqui
6
7  // Este código abaixo já havia sido feito anteriormente
8  @property (weak, nonatomic) IBOutlet UITextField *nomeField;
9  @property (weak, nonatomic) IBOutlet UITextField *quantidadeField;
10 @property (weak, nonatomic) IBOutlet UILabel *avisoSucessoLabel;
11 - (IBAction)incrementadorAlterado:(id)sender;
12 - (IBAction)salvar:(id)sender;
13
14 @end
```

O código referente a esta seção é o da linha 4, lembrando que é necessário adicioná-lo entre chaves (final da linha 3 e linha 5), como foi explicado anteriormente na seção da classe `Empresa`. Da linha 8 em diante é o código que criamos anteriormente, e foi adicionado na listagem apenas para referência.

Voltando o arquivo `ViewController.m` (lembra da tecla de atalho?), vamos criar o método para armazenar as empresas no objeto `catalogo`, e depois chamar o método na ação do botão "Salvar". Adicione o seguinte código logo antes do método `- (IBAction)salvar:(id)sender`:

```
1 -(void) salvaEmpresa:(Empresa *) novaEmpresa {
2     if (!catalogo) {
3         catalogo = [[NSMutableArray alloc] init];
4     }
5
6     [catalogo addObject:novaEmpresa];
7 }
```

Na linha 6 adicionamos na lista o objeto `novaEmpresa` passado como argumento ao método. Já as linhas 2 e 3 precisam de uma explicação mais detalhada: em Objective-C objetos não-instanciados contêm por padrão o valor `nil` (que é um primo próximo do `NULL` de Java, C# e do próprio C e C++). Além disso, na linguagem os valores `nil` e `0` (zero) são tratados como "falso" em expressões condicionais,

da mesma forma que *qualquer coisa* que não seja `nil` ou `0` é considerada como "verdadeiro" em condicionais booleanas. Em outras palavras, os seguintes códigos são equivalentes:

```
// OK
if (!catalogo) {

}

// OK - mesmo efeito do exemplo anterior
if (catalogo == nil) {

}
```

Por fim, na linha 3 criamos uma nova instância de fato do `NSMutableArray`

NSArray versus NSMutableArray - imutável e mutável

Algumas classes do SDK do iOS contêm versões imutáveis (que após criadas não podem mais ser modificadas) e versões mutáveis (que podem ser modificadas livremente), como é o caso do `NSArray` (imutável) e `NSMutableArray` (mutável). Saber quando usar uma versão ou outra depende exclusivamente das necessidades de cada parte do código. Outras classes imutáveis que têm versões mutáveis são `NSDictionary`, `NSSet`, `NSData` e `NSString`.

3.14 Listando todas as empresas do catálogo

Aproveitando que estamos embalados, vamos criar também um outro método que lista todas as empresas adicionadas ao `catalogo`. Adicione o seguinte código logo após o método `salvaEmpresa`:

```
-(void) mostraCatalogo {
    NSLog(@"******* Listando todas empresas *******");

    for (Empresa *empresa in catalogo) {
        NSLog(@"A empresa %@ tem %d funcionários",
            empresa.nome, empresa.quantidadeFuncionarios);
    }
}
```

Modifique o método `salvar:` para remover a antiga chamada a `NSLog`, e invoque os dois novos métodos que criamos. Confira abaixo:

```
- (IBAction)salvar:(id)sender {
    Empresa *e = [[Empresa alloc] init];
    e.nome = self.nomeField.text;
    e.quantidadeFuncionarios = [self.quantidadeField.text intValue];

    [self salvaEmpresa:e];
    [self mostraCatalogo];
}
```

Rode novamente o aplicativo no simulador através da tecla de atalho `Command + R` (ou pelo menu... você ainda lembra qual é?) e insira algumas empresas, para ver o resultado do aplicativo até o momento.

3.15 Vença a sintaxe do Objective-C: invocação de métodos

A assinatura de métodos em Objective-C é algo bastante particular da linguagem, e pode ser um pouco criptográfico numa primeira olhada. Uma das intenções dos criadores da linguagem foi criar uma sintaxe declarativa e fácil de ler, que não deixasse margem para dúvidas em relação ao que se refere cada valor passado ao método. Se isso é algo bom ou ruim, cabe a você decidir.

Considere o seguinte código, retirado da classe `NSMutableArray`:

```
- (void) insertObject:(id) objeto atIndex:(NSInteger) indice;
```

A declaração do método começa com o tipo de acesso, seguido do tipo de retorno e do nome do método. Se o método começa com o sinal de soma (+) significa que ele é um método de classe (que é algo próximo ao `static` do Java e C#), e se começar com o sinal de subtração (-), é um método de instância. Veremos mais sobre isso no capítulo sobre Objective-C avançado. O *retorno* pode ser qualquer tipo válido de Objective-C ou C, como `int`, `float`, `id`, `NSInteger`, `void` etc.

Para passar **um** argumento para o método é bastante simples, bastando utilizar o sinal de dois pontos (":") para separar o nome do método da variável. Por exemplo, o código abaixo insere uma empresa em uma posição específica:

```
[catalogo insertObject:empresa atIndex:3];
```

Toda invocação de método precisa ser feita entre colchetes, aninhando-os conforme necessário (ou seja, colchete dentro de colchete dentro de colchete). Sim, o código fica um pouco difícil de ler, em especial enquanto você está aprendendo mais da linguagem.

Já um método que recebe múltiplos argumentos pode assustar um pouco mais (pelo menos assustou a mim, por semanas), porém compreender sua estrutura é fundamental. Considere o seguinte método, também de `NSMutableArray`, que serve para substituir uma parte dos elementos com objetos de outro `NSArray`:

```
- (void)replaceObjectsInRange:(NSRange) range
    withObjectsFromArray:(NSArray *) outroArray
    range:(NSRange) rangeDoOutroArray
```

Para efeitos de comparação, o mesmo método em Java seria declarado parecido com isso::

```
void replaceObjectsInRange(NSRange range, NSArray outroArray,
    NSRange rangeDoOutroArray);
```

Para visualizar melhor a formação da *assinatura* do método, veja a figura 3.11

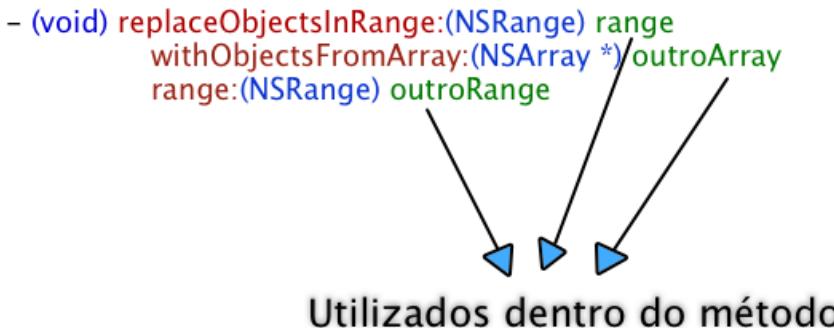



Figura 3.11: Estrutura da assinatura de um método em Objective-C

Note que, embora assustador em um primeiro momento, o padrão de construção para os demais argumentos se repete, sendo formado sempre por 3 partes: a palavra que fará parte do *nome* do método, o tipo da variável e o nome da variável em si.

É esta última que será utilizada dentro do método. Muitas vezes, a palavra que faz parte do *nome* do método e a variável costumam ser iguais, mas isso não é uma regra.

A nomenclatura irá variar de acordo com cada caso, sendo que o mais importante é que você utilize nomes que façam sentido ao contexto de uso. Considere o exemplo abaixo, de um método que recebe o nome e quantidade de funcionários de uma empresa, e retorna uma instância da classe:

```
-(Empresa *) criaEmpresaComNome:(NSString *) nome
    comNumeroDeFuncionarios:(int) quantidade {

    Empresa *e = [[Empresa alloc] init];
    e.nome = nome;
    e.quantidadeFuncionarios = quantidade;
    return e;
}
```

Para invocar, fazemos:

```
Empresa *novaEmpresa = [self criaEmpresaComNome:@"Jujubas LTDA"
    comNumeroDeFuncionarios:3];
```

A verbosidade do nome dos métodos tem um motivo especial, que é fazer com que o código seja fácil de ler e expresse bem o domínio no qual é aplicado.

3.16 Criando instâncias de objetos

Para poder criar instâncias de classes em Objective-C são necessários dois passos: alocar memória, e iniciar o objeto. Estes passos geralmente são feitos em conjunto, e o objeto somente poderá ser utilizado depois que estes dois passos tiverem sido completados com sucesso. Também não existe o conceito de *construtores* propriamente ditos, mas sim de *inicializadores*, que por padrão chama-se `init`. Veja o código abaixo:

```
Empresa *e = [[Empresa alloc] init];
```

A primeira parte do processo de instanciação ocorre com o método `alloc`, o qual reserva memória necessária para todas as variáveis de instância e as inicia com os valores padrão. Em seguida, o método `init` prepara a instância de fato, permitindo que seja utilizada. O método `init` retorna um tipo `id`, que é como um "ponteiro para qualquer objeto"; pense nele como sendo um primo do `Object`

de Java e C#, contudo *muito* mais abrangente (existe o tipo `NSObject`, porém). `init` é disponibilizado por padrão, não sendo necessário que você implemente-o toda vez. Entretanto, caso queira fazer uma rotina de inicialização do objeto, basta sobrescrevê-lo:

```
1 -(id) init {
2     self = [super init];
3 
4     if (self) {
5         // Se chegou aqui, a inicialização ocorreu com sucesso
6     }
7 
8     return self;
9 }
```

A linha 2 invoca o método `init` na classe-pai (que será pelo menos `NSObject`), enquanto a linha 4 verifica se o processo ocorreu com sucesso. Caso tenha falhado, `self` irá conter o valor `nil`.

É possível criar inicializadores customizados, exatamente como faríamos como qualquer outro método, prestando atenção apenas na necessidade de chamar algum outro inicializador da classe-pai (ou um outro na mesma classe, que eventualmente chame o da classe-pai). Por exemplo, poderíamos criar um inicializador para a classe `Empresa` que já recebe o nome e a quantidade de funcionários:

```
-(id) initWithNome:(NSString *) nome eQuantidadeFuncionarios:(int)
                                                        quantidade {
    if ((self = [super init])) {
        self.nome = nome;
        self.quantidadeFuncionarios = quantidade;
    }

    return self;
}
```

Para criar uma nova instância:

```
Empresa *ep = [[Empresa alloc] initWithNome:@"Jujubas LTDA"
    eQuantidadeFuncionarios:17];
```

3.17 Melhoria: esconder o teclado automaticamente

Um problema com o nosso aplicativo é que, uma vez que o teclado aparece, ele nunca mais some, nem mesmo após salvarmos a empresa. Além disso, a imagem de sucesso fica escondida. Isso ocorre porque em iOS não existe exatamente o conceito de "foco" nos componentes como temos em aplicações tradicionais ou mesmo na Web, mas sim algo chamado de *responder*, que são objetos que podem lidar com diversos tipos de eventos. No caso de componentes como o `UITextField` que utilizamos no aplicativo deste capítulo, eles aceitam as mensagens de entrada de texto (que disparam o teclado), porém outros componentes como botões (o `UIButton` no nosso caso) ignoram tais eventos. Então, o componente que inicialmente pegou o teclado fica com ele até que outro componente o peça ou, então, que explicitamente recebam uma mensagem para liberá-lo. E é isso o que faremos.

> **A cadeia de eventos**
>
> Tecnicamente os motivos são um pouco mais densos, e na explicação desta seção optamos por apresentar de uma forma mais curta e simples para não sair do foco do capítulo. Para saber mais a respeito, procure por "*UIResponder Chain*" na Internet.

No método `salvar:` adicione uma chamada ao método `resignFirstResponder` do componente de nome da empresa, conforme o código abaixo:

```
1 - (IBAction)salvar:(id)sender {
2     // Libera o teclado
3     [self.nomeField resignFirstResponder];
4
5     // Restante do método
6 }
```

O código da linha três diz algo como "eu me abstenho de responder pelos eventos por enquanto", fazendo com que o teclado desapareça. Se quiser forçar o teclado a aparecer, basta usar o método `becomeFirstResponder`.

3.18 Melhoria: mostrando a mensagem de sucesso somente ao salvar

Até o momento a mensagem "Dados salvos com sucesso" aparece o tempo todo, quando na prática deveria aparecer após adicionarmos uma nova empresa, e ainda assim apenas por alguns momentos. Para solucionar isso primeiro precisamos escondê-la ao iniciar o aplicativo, e mostrá-la na ação do botão "Salvar".

Para a primeira primeira parte, atribua o valor `YES` à propriedade `hidden` da variável `avisoSucessoLabel`, isso tudo no método `viewDidLoad` do arquivo `ViewController.m`, conforme o código abaixo:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.avisoSucessoLabel.hidden = YES;
}
```

Em tempo: todo componente tem a propriedade `hidden`. Para a segunda parte — mostrar a mensagem por alguns instantes — vamos novamente modificar o método `salvar:`. Adicione o código abaixo ao final do método:

```
1  - (IBAction)salvar:(id)sender {
2      // Código já existente omitido apenas no livro
3
4      self.avisoSucessoLabel.alpha = 0;
5
6      [UIView animateWithDuration:1 animations:^{
7          self.avisoSucessoLabel.hidden = NO;
8          self.avisoSucessoLabel.alpha = 1;
9      } completion:^(BOOL finalizado) {
10         [UIView animateWithDuration:1 delay:2 options:0 animations:^{
11             self.avisoSucessoLabel.alpha = 0;
12         } completion:^(BOOL finalizado) {
13             self.avisoSucessoLabel.hidden = YES;
14         }];
15     }];
16 }
```

Se você se assustou com este último pedaço de código, não se preocupe: você não é o único. Objective-C pode ser um tanto amedrontador em certos momentos. Além disso, o código ficou um pouco maior porque estamos fazendo duas animações: a primeira para mostrar o aviso, e a segunda para escondê-la novamente após

um certo tempo. Mais adiante, no capítulo sobre `UIViews` mostraremos a fundo o funcionamento dos blocos de animação, que são um recurso mais avançado da linguagem.

Criação do bloco de animação

Repare que o bloco de animação criado na última listagem de código utiliza o caracter circunflexo (`^`) para indicar o início de um pedaço de código. Em alguns teclados é necessário apertar a tecla que contêm este símbolo, seguido da barra de espaço.

Rode o aplicativo, insira uma empresa, e você verá a mensagem aparecendo de maneira animada, e depois de 2 segundos tornando a desaparecer automaticamente. Você está ficando bom nisso!

Capítulo 4

Coordenando o trabalho com controladores

Diferentes plataformas contêm diferentes maneiras de resolver uma necessidade bastante comum, que é a de gerenciar diversas telas e funcionalidades. Além disso, é necessário que a navegação e transição entre elas sejam uma tarefa intuitiva para o usuário e lógica para o desenvolvedor. Em aplicativos iOS isso é feito com o uso de controladores (*controllers*), que são classes que — como o próprio nome sugere — controlam (ou gerenciam) um conjunto de funcionalidades diretamente relacionadas.

Controladores (do inglês "*controllers*") são classes normais, e portanto podem conter qualquer tipo de lógica e regra de negócios, embora — ao menos teoricamente — o ideal é que sejam usados como um intermediário, como um facilitador de trabalho entre a camada visual (as *views*) e a lógica de negócios. Na prática, muitos desenvolvedores acabam misturando as coisas em um único lugar, o que pode levar a dificuldades para manter e evoluir o aplicativo a longo prazo.

O código-fonte deste capítulo está disponível nas pastas "ViewControllerAnimations" e "NavigationControllerDemo" (lembrando que o endereço do site com os códigos está na introdução do livro).

O objetivo deste capítulo é mostrar diferentes maneiras de interagir entre `View Controllers`, que são as classes responsáveis por coordenar o trabalho em aplicações iOS, tais como redimensionamento de *views*, eventos de rotação do dispositivo e navegação entre as diversas partes do aplicativo. A figura 4.1 mostra o papel de um controlador frente a diversas outras partes de um aplicativo.

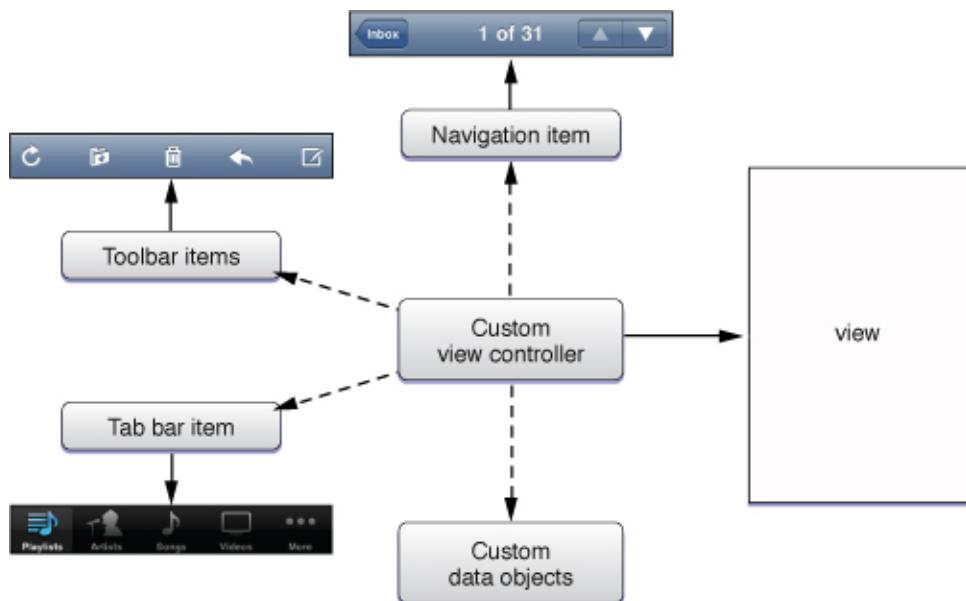



Figura 4.1: Papel do controlador. Fonte: Apple

Controladores por si só não são componentes visuais. Pense neles como um esqueleto, que embora não possa ser visto, é vital para que tudo se encaixe. A interface com o usuário é feita através de *views*, tanto que ele tem a sua própria propriedade chamada `view`.

Controladores e Storyboards

Todo o conteúdo apresentado neste capítulo é fundamental para o correto entendimento de um dos principais pilares de aplicações iOS, e tem direta ligação com o conteúdo do próximo capítulo, 5.

4.1 Passando de um controlador para outro

Existem diversas formas de fazer a transição entre controladores, como apresentá-los de forma *modal*, ou então utilizando um navegador especializado. Veremos essas duas abordagens neste capítulo.

Controladores do tipo *Modal* funcionam da mesma maneira como *janelas modais* em aplicativos desktop, que é aquele comportamento no qual, quando um componente desses é apresentado, nada do componente anterior pode ser acessado enquanto este não for fechado.

Crie um novo projeto do tipo *Single View Application* (File -> New -> Project -> Single View Application) chamado " `ViewControllerAnimations`" e selecione *iPhone* como Device. O Xcode irá criar um projeto com os arquivos `AppDelegate.m` e `.h`, e outra classe chamada `ViewController.m` e `.h`, além do arquivo `Main.storyboard`.

Abra o arquivo `Main.storyboard`, e adicione quatro componentes do tipo *Button*, com os textos "*Dissolver*", "*Virar página*", "*Subir vertical*" e "*Girar horizontal*" — estas serão as operações que realizaremos.

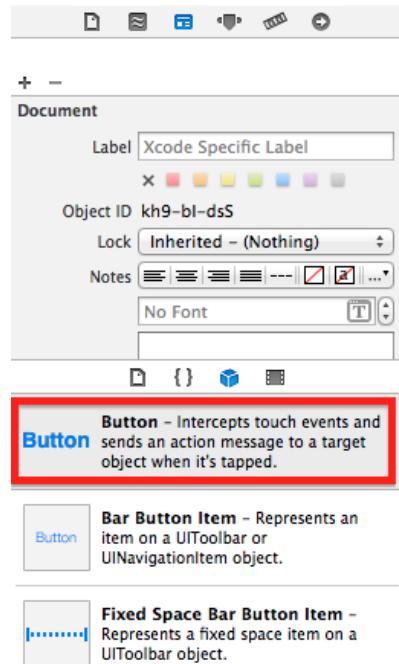



Figura 4.2: Localização do Round Rect Button, no Xcode

A tela deverá ficar como a da figura 4.3.

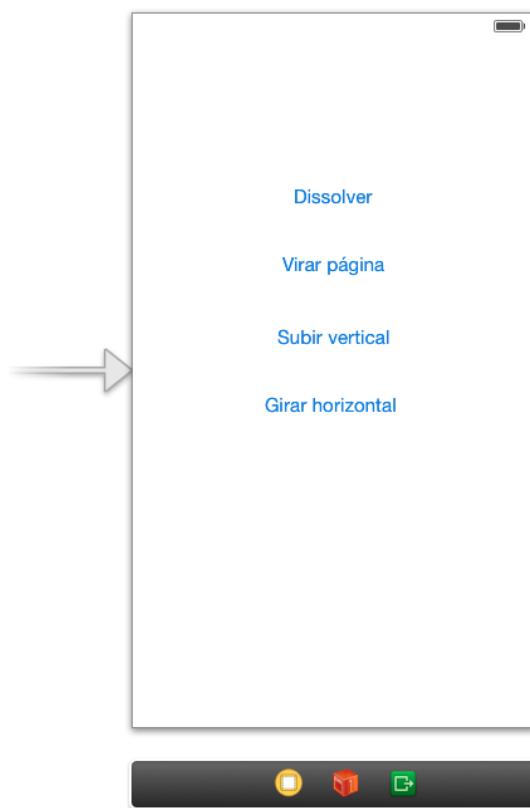



Figura 4.3: Tela com os botões posicionados

O próximo passo é conectar as ações destes botões. Abra o *Assistant Editor* (`View -> Assistant Editor -> Show Assistant Editor`, ou `Option + Command + Enter`), e conecte uma ação para cada um dos botões no arquivo “ `ViewController.h`”, da seguinte forma:

- Texto: “Dissolver”, nome do método: “showDissolve”
- Texto: “Virar página”, método: “showPageCurl”
- Texto: “Subir vertical”, método: “showVertical”
- Texto: “Girar horizontal”, método: “showHorizontal”

O código do arquivo `ViewController.h` deverá estar como mostrado abaixo:

```
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController
- (IBAction)showDissolve:(id)sender;
- (IBAction)showPageCurl:(id)sender;
- (IBAction)showVertical:(id)sender;
- (IBAction)showHorizontal:(id)sender;
@end
```

Cada um dos métodos que iremos implementar terá como tarefa mostrar outro controlador ao usuário, sobrepondo o principal temporariamente. A forma que veremos é uma das várias possíveis maneiras, e mais adiante será mostrado como utilizar a classe `UINavigationController` para fazer a navegação entre eles de maneira organizada e gerenciável.

Para que os botões possam abrir um novo controlador é necessário primeiro criá-los. Para tanto, acesse o menu `File -> New -> File...` e selecione o template *Objective-C class*, e clique no botão Next. Na próxima tela devemos informar o nome da classe a ser criada, e qual será a classe pai. Chame-a de "OpcoesController", e em "*Subclass of*" selecione a opção "UIViewController". Marque também o checkbox "With XIB for user interface", conforme mostrado na figura 4.4.

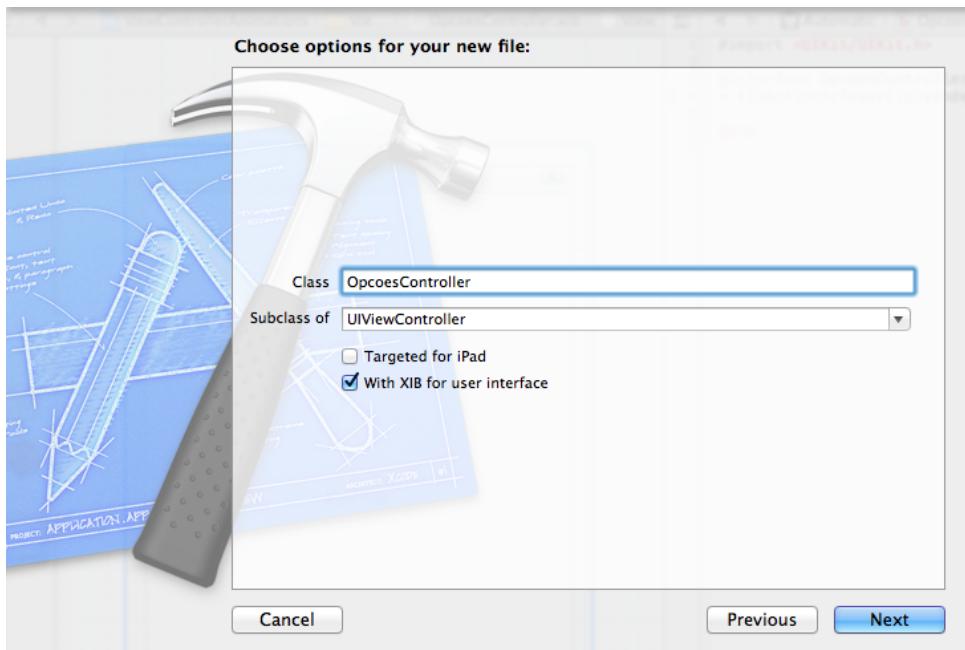



Figura 4.4: Criando o controlador para a tela de opções

Quando o Xcode perguntar onde o arquivo deverá ser salvo, selecione a pasta raiz do projeto, no mesmo lugar onde estão os outros arquivos. Você deverá ter o arquivo "OpcoesController" nas extensões ".h", ".m" e ".xib".

Reparem bem que todos os arquivos têm o mesmo nome, mudando a extensão. Embora não seja obrigatório nomear deste jeito, esta é a maneira que todos os desenvolvedores costumam utilizar. Além disso, quando trabalhamos com interfaces gráficas utilizando arquivos `.xib`, respeitar a convenção de nomes simplifica o código, pois o comportamento padrão do iOS é procurar o `.xib` que tenha o mesmo nome da classe.

Abra o arquivo "`OpcoesController.xib`" e adicione um componente do tipo "*Label*" no meio da tela, com o texto "Opções", apenas para termos uma indicação visual quando ele for exibido.

Agora vá ao arquivo `ViewController.m`, onde você irá encontrar o corpo dos métodos que foram conectados anteriormente (pelo `Main.storyboard`) ao arquivo `ViewController.h`. Cabe a nós implementar cada uma das ações dos botões. O primeiro método é o `showVertical`, que representa a animação padrão

do iOS para o tipo de operação que iremos realizar. O que deve ser feito é criar uma instância do `OpcoesController`, especificar o tipo da animação e invocar o método `presentModalViewController`.

Vamos primeiro ver a implementação completa do método, conforme a listagem abaixo:

```
1 - (IBAction)showVertical:(id)sender {
2     OpcoesController *c = [[OpcoesController alloc] init];
3     c.modalTransitionStyle = UIModalTransitionStyleCoverVertical;
4 
5     [self presentViewController:c animated:YES completion:nil];
6 }
```

Obs: não esqueça de adicionar a instrução `#import "OpcoesController.h"` no início do arquivo.

Na linha 3 é definido o tipo de animação desejado, que no caso é `UIModalTransitionStyleCoverVertical`. Esta é a animação padrão, e podemos omitir isso na prática. Porém, como neste caso desejamos testar todas as animações possíveis, é interessante atribuirmos explicitamente o tipo desejado.

Já a linha 5 é a responsável por mostrar de fato o nome controller ao usuário, através do método `presentViewController:animated:completion`.

Rode o aplicativo (`Command+R`) e clique no botão "*Subir vertical*". Você deverá ver o `OpcoesController` cobrir toda a tela, vindo de baixo para cima. Esta foi a primeira animação. O próximo método a implementar é o `showHorizontal`, que mostrará utilizando um efeito de girar a tela horizontalmente. A implementação completa está abaixo:

```
- (IBAction)showHorizontal:(id)sender {
    OpcoesController *c = [[OpcoesController alloc] init];
    c.modalTransitionStyle = UIModalTransitionStyleFlipHorizontal;

    [self presentViewController:c animated:YES completion:nil];
}
```

Rode novamente o aplicativo e clique no botão "Girar horizontal", e veja a diferença no efeito da animação. Bacana, não?

Repare bem na implementação dos métodos `showVertical` e `showHorizontal`, e veja que a maior parte do código deles é igual, mudando apenas o tipo de animação desejada. Como temos mais duas animações

para implementar, o ideal seria isolar o código em comum a todos os métodos e centralizá-lo em um único lugar, evitando assim a duplicação de trabalho. Embora estejamos fazendo apenas um aplicativo de testes, quanto mais cedo adotar a prática de reutilizar código, mais rapidamente ela se tornará natural, e consequentemente o seu sistema ficará mais fácil de manter.

Portanto, antes de implementar os demais métodos, vamos primeiro criar um método que fará todo o trabalho repetitivo, recebendo como parâmetro o tipo de animação a ser utilizada. Veja a implementação completa abaixo:

```
-(void) mostraControllerComAnimacao:(UIModalTransitionStyle) estilo {
    OpcoesController *c = [[OpcoesController alloc] init];
    c.modalTransitionStyle = estilo;

    [self presentViewController:c animated:YES completion:nil];
}
```

O método `mostraControllerComAnimacao:` é praticamente igual ao código que fizemos anteriormente, com a diferença de que ele recebe um argumento especificando o tipo de animação. Coloque a implementação dele em qualquer lugar da classe (por exemplo, logo abaixo de `@implementation`), e implemente o resto dos métodos da seguinte maneira:

```
- (IBAction)showDissolve:(id)sender {
    [self
       mostraControllerComAnimacao:UIModalTransitionStyleCrossDissolve];
}

- (IBAction)showPageCurl:(id)sender {
    [self
        mostraControllerComAnimacao:UIModalTransitionStylePartialCurl];
}

- (IBAction)showVertical:(id)sender {
    [self
       ostraControllerComAnimacao:UIModalTransitionStyleCoverVertical];
}

- (IBAction)showHorizontal:(id)sender {
    [self
      mostraControllerComAnimacao:UIModalTransitionStyleFlipHorizontal];
}
```

Bem mais simples e prático, e sem duplicar código. Rode novamente o aplicativo e teste cada um dos botões.

4.2 Fechar um controlador modal

Uma coisa que você deve ter notado é que, depois de abir um controlador usando `presentViewController`, não havia nenhuma maneira de fechá-lo ou voltar para a tela anterior. Contudo, isso é bastante simples de se resolver através do método `dismissViewControllerAnimated:completion`, que deve ser adicionado como ação de um botão na classe `OpcoesController`. Siga os seguintes passos:

Abra o arquivo `OpcoesController.xib` e adicione um botão com o texto "Fechar", e conecte uma *action* chamada "close" no arquivo `OpcoesController.h` (lembre-se de usar o *Assistant Editor* para ver o arquivo `.xib` e `.h` lado a lado). Em seguida, abra o arquivo `OpcoesController.m` e chame o método `dismissViewControllerAnimated:completion`, conforme abaixo:

```
- (IBAction)close:(id)sender {
    [self dismissViewControllerAnimated:YES completion:nil];
}
```

Rode novamente o aplicativo "clique" no botão "Fechar", para que o controlador seja fechado.

4.3 Navegar por diferentes telas com o UINavigationController

Uma das formas de navegação mais comum em aplicativos iOS é aquela em que, feita uma determinada ação, uma nova tela desliza da direita para a esquerda por cima da anterior. Depois, através de um toque em um botão no canto superior esquerdo (geralmente um botão escrito "*Back*" ou "*Voltar*"), volta-se para a tela anterior. Ou seja, é uma navegação hierárquica.

Este tipo de navegação chama-se "Controlador de Navegação", ou no termo técnico, `UINavigationController`. Nos exemplos iremos nos referir a este componente pelo nome *navigation controller*, pois é um termo bastante conhecido em desenvolvimento iOS, e acostumar-se com ele facilitará a busca por informações na Internet, posteriormente. A figura 4.5 mostra como ele funciona. Ela deve permitir

que troquemos de uma tela para outra, de uma funcionalidade para outra, através de uma interface simples.

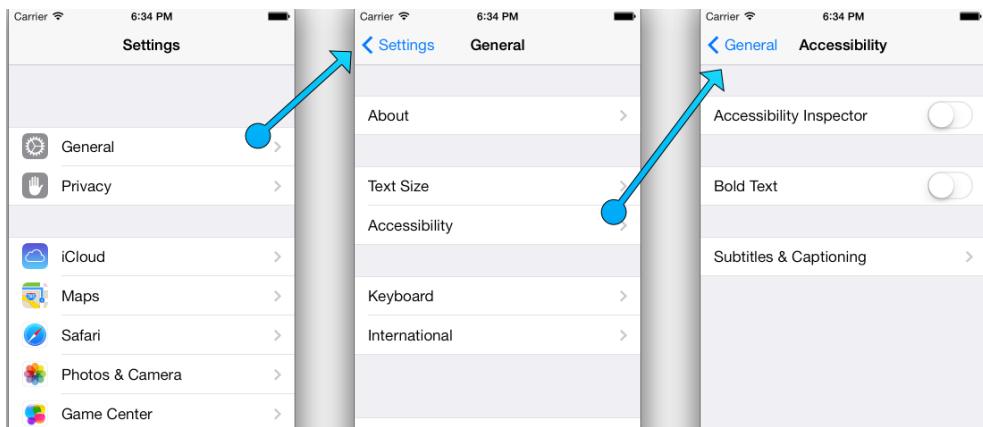
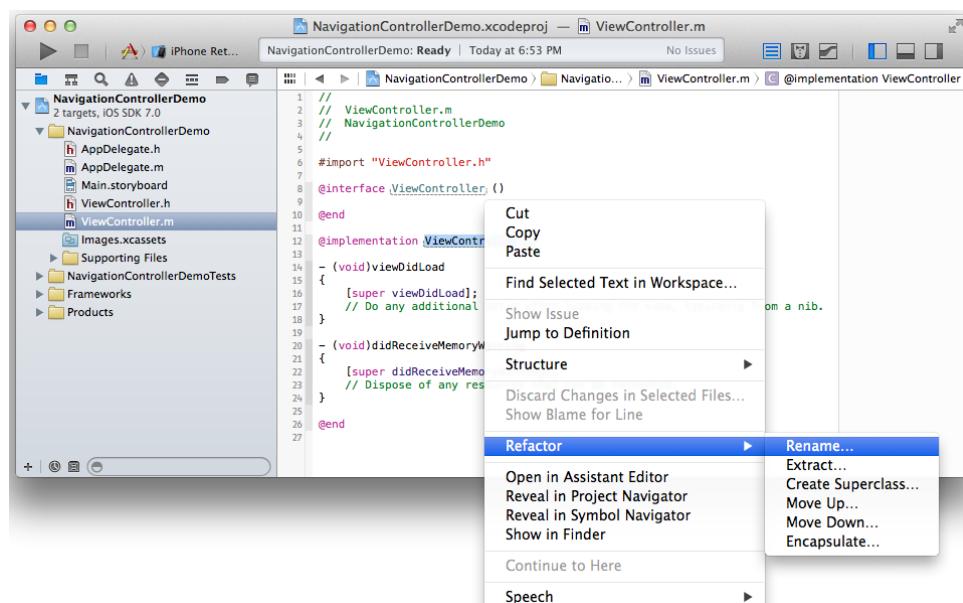



Figura 4.5: Funcionamento de um UINavigationController

Navigation controllers são peças muito importantes em aplicações iOS, pois permitem a transição de um controller para outro sem que haja perda de estado. Em outras palavras, o aplicativo literalmente vai empilhando os controladores, sem tirá-los da memória, permitindo que a navegação reversa seja feita de maneira natural.

Gerenciamento de memória de controladores

Em condições normais de funcionamento, os controladores que estão na hierarquia de um Navigation Controller são sempre mantidos em memória. Porém, não devemos trabalhar com a premissa de que isso será sempre verdade, pois no caso de falta de memória, o iOS pode "descarregar" alguns deles, recriando-os quando necessário.

Para entender melhor o funcionamento do `UINavigationController`, vamos criar um aplicativo de um catálogo de empresas, no qual é possível adicionar novos registros e realizar configurações.

Crie um novo aplicativo do tipo "Single View Application", nos mesmos moldes dos exemplos anteriores, e chame-o de "NavigationControllerDemo". Você deverá ter um projeto padrão, com o `AppDelegate` e uma classe chamada

`ViewController`. Repare que este nome, “ViewController”, que o Xcode cria junto com o projeto, é apenas para nossa conveniência, porém você não precisa ficar amarrado a ele.

Para começar, vamos mudar o nome da classe “ViewController” para algo semanticamente melhor. Uma das maneiras de fazer isso é renomear manualmente os arquivos `ViewController.m` e `.h`, porém isso dá muito trabalho e corremos o risco de esquecer alguma coisa. Uma maneira mais inteligente é utilizar a funcionalidade de *refatoração* do Xcode, que faz para nós todo trabalho pesado.

Abra o arquivo `ViewController.m` e, na linha de definição da classe (onde tem `@implementation ViewController`) clique com o botão direito e selecione a opção `Refactor -> Rename...`, conforme a figura 4.6.

Na caixa de diálogo que abrir, informe o valor “*RootController*” no campo de texto, e deixe marcada a opção “Rename related files”, para que o Xcode renomeie todos os arquivos que tenham relação ao original `ViewController`.

Ao clicar no botão “Preview”, o Xcode irá mostrar um resumo das operações que serão realizadas. Clique no botão “Save”, e caso o Xcode mostre uma mensagem perguntando se deve fazer backup do conteúdo, clique em “Enable”.

Ao término da operação, repare que tanto os arquivos em disco como as próprias definições da classe `ViewController` foram renomeadas para `RootController`.

4.4 Preparar a tela principal

Agora precisamos colocar botões no `RootController` para chamar os outros controladores, um para a fictícia tela de adicionar empresas ao catálogo, e outro para a — adivinhem — também fictícia tela de configurações. O objetivo principal desta seção é demonstrar como navegar de um lugar para o outro, e não na implementação real de um catálogo propriamente dito.

Abra o arquivo `Main.storyboard` e coloque 2 botões, o primeiro como texto "Adicionar", e o segundo com o texto "Configurações", conforme mostra a figura 4.6.

Figura 4.6: Desenho da tela principal do aplicativo

Por que editar o arquivo Main.storyboard?

O motivo pelo qual, neste ponto, é necessário editar o arquivo `Main.storyboard`, se deve ao fato que ele é a forma padrão pela qual o Xcode cria projetos do tipo "*Single View Application*" (além de alguns outros), sendo o ponto de partida da app. Esta é também uma forma da Apple estimular o uso de Storyboards em detrimento do antigo formato que se baseada apenas em arquivos " `.xib`". Somado a isso, você deve se lembrar que todo novo projeto também inicia com a classe " `ViewController`", que anteriormente renomeamos para " `RootController`", e esta mesma classe também está configurada como sendo a primeira que o aplicativo deverá utilizar, quando o aplicativo for iniciado.

No capítulo 5 veremos mais a fundo este assunto.

4.5 Conectar as ações dos botões

Da mesma forma como já fizemos diversas outras vezes durante o livro, para que os botões tenham utilidade é necessário conectar as ações deles ao código, utilizando a já conhecida abordagem "Selecionar componente -> Arrastar com CTRL para o arquivo `.h` -> *Connection* do tipo *Action*". Mais fácil que devorar o bolo de chocolate da vovó. Chame a ação do botão "Adicionar" de " `abrirAdicionar`", e para segundo botão nomeie a ação de " `abrirConfiguracoes`". O seu arquivo `RootController.h` deverá ficar assim:

```
#import <UIKit/UIKit.h>

@interface RootController : UIViewController
- (IBAction)abrirAdicionar:(id)sender;
- (IBAction)abrirConfiguracoes:(id)sender;

@end
```

4.6 Criar a tela de Adicionar Empresa

Adicione uma nova classe (`Command + N`) chamada `AdicionarController` da mesma forma como criamos o `RootController`, lembrando de selecionar a opção

para criar o arquivo `.xib`, e adicione apenas um *label* com o texto "*Tela adicionar empresa*" no arquivo `AdicionarController.xib`. Lembre-se que o objetivo é focar na navegação entre os controladores, e não na funcionalidade de adicionar empresas propriamente dita.

4.7 Navegar de um controlador para outro

A classe `UIViewController` contém uma propriedade chamada `navigationController` que representa a instância do `UINavigationController` utilizada pelo aplicativo, sendo que os principais métodos são dois:

- `pushViewController:animated`, para navegar para um outro controlador
- `popViewControllerAnimated`, para voltar um nível na hierarquia.

Para ir do `RootController` para a tela de adicionar empresa, implemente o método `abrirAdicionar` no arquivo " `RootController.m`" da seguinte forma (obs: lembre-se de importar o arquivo `AdicionarController.h`):

```
- (IBAction)abrirAdicionar:(id)sender {
    AdicionarController *c = [[AdicionarController alloc] init];
    [self.navigationController pushViewController:c animated:YES];
}
```

Rode o aplicativo (`Command+R`) e interaja com o botão "Adicionar". Aconteceu alguma coisa? Muito provavelmente não, pois apesar da propriedade `navigationController` fazer parte da API da classe `UIViewController`, o valor dela não é criado automaticamente pelo iOS. Portanto, é nossa responsabilidade disponibilizar um `UINavigationController` e instruir o aplicativo a utilizá-lo.

4.8 Associar um UINavigationController ao projeto

Conceitualmente um `UINavigationController` não deve ser estendido, mas sim ser utilizado diretamente. Imagine o navigation controller como sendo um *gerenciador*, que recebe uma referência para o controlador inicial, e a partir dai permite a navegação para os outros através do método `pushViewController:animated`.

A propriedade `navigationController` da classe `UIViewController` é vazia em controllers que não façam parte da hierarquia de um navigation controller (o que faz bastante sentido). Porém, a partir do momento em que criamos um `UINavigationController`, o iOS atribui automaticamente um valor válido para a propriedade do controlador principal e de todos os outros que fizerem parte da navegação. Isso é bastante prático, pois não precisamos nos preocupar em manter uma referência global ao navigation controller e atribuí-lo manualmente aos demais.

Para adicionar um navigation controller ao projeto, abra o arquivo `Main.storyboard` e selecione a barra preta na parte inferior, conforme mostra a imagem 4.7, e em seguida acesse o menu " `Editor -> Embed In -> Navigation Controller`". O resultado ficará como o da imagem 4.8.

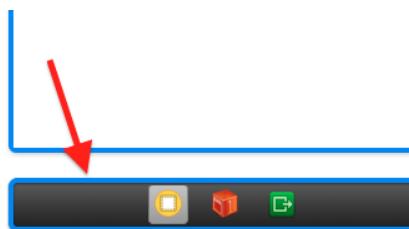



Figura 4.7: Clique na barra para selecionar o controlador

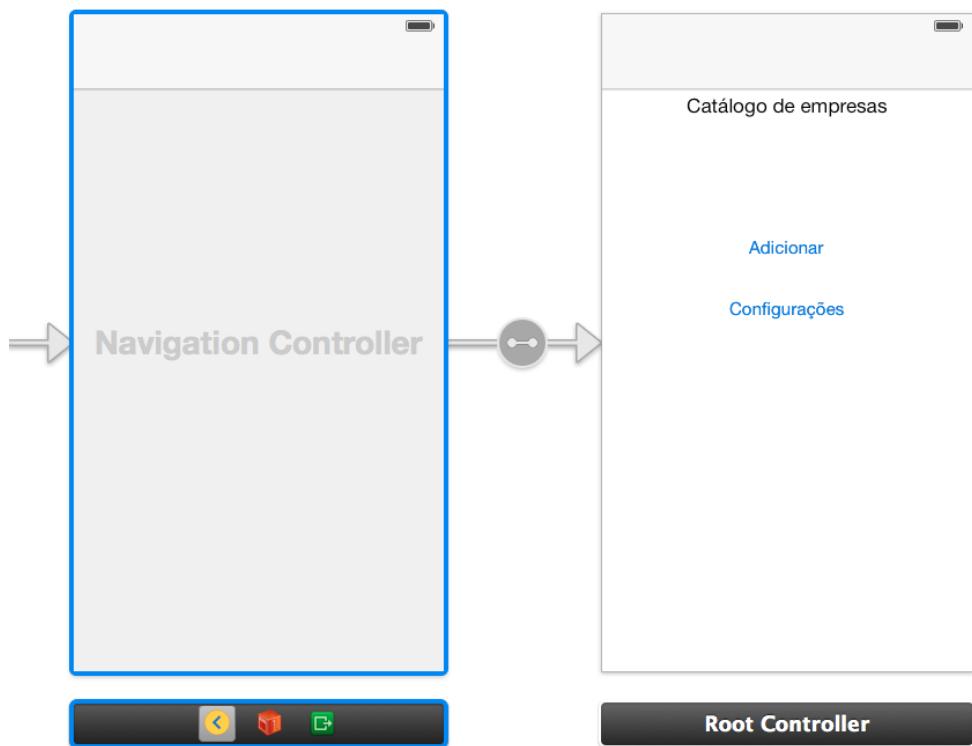



Figura 4.8: Navigation Controller associado ao projeto

Rode novamente o aplicativo e interaja com o botão "Adicionar", e veja que desta vez a tela do `AdicionarController` é exibida corretamente. Uma outra coisa nova é que o aplicativo apresenta uma barra superior de navegação — chamada de "Navigation Bar" —, onde um botão com o título "Back" aparece no canto superior esquerdo quando navegamos de um lugar para o outro. Tecnicamente funciona desta maneira: ao executarmos `pushViewController:animated`, o iOS usará o título do controlador *anterior* (representado pela propriedade `title`) como texto do botão e, caso não haja nenhum título definido, o texto "Back" é utilizado.

4.9 Criar os demais controladores

O nosso aplicativo também precisa de uma hipotética tela de configurações — aliás, duas. A primeira delas contém algumas coisas gerais, como se o catálogo deve ser utilizado em "Modo seguro" e se os dados devem ser salvos automaticamente ou

não. Além disso, precisamos dos dados de acesso do usuário. Para o primeiro caso crie um novo controlador chamado `ConfguracoesController` como feito anteriormente, e adicione alguns componentes para parecer que ele tem utilidade, como exemplificado na figura 4.9.

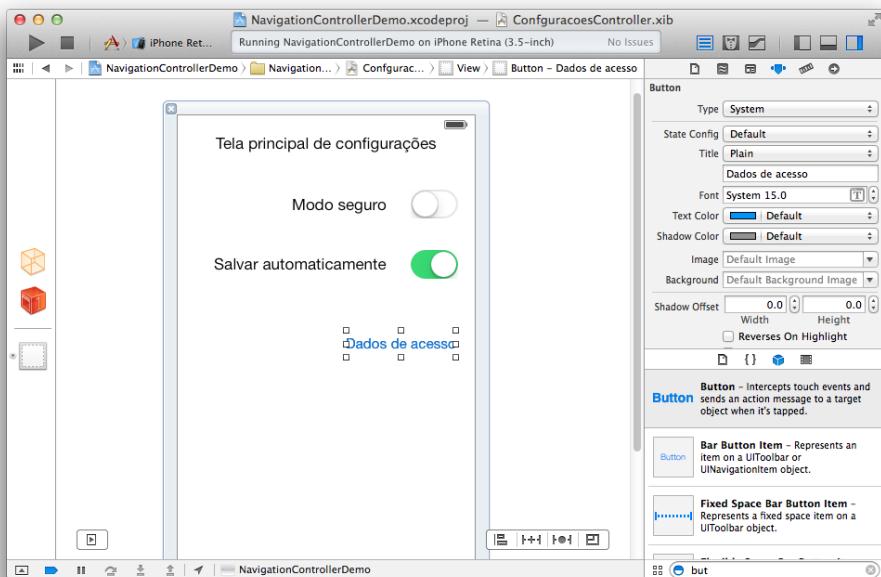



Figura 4.9: Exemplo da tela principal de configurações

IMPLEMENTE O MÉTODO `ABRIRCONFIGURACOES:`

Não esqueça de implementar o corpo do método `abrirConfiguracoes:` no arquivo `RootController.m`, seguindo a mesma lógica do que foi feito para o método `abrirAdicionar:`, porém desta vez instanciando o `ConfiguracoesController`.

Além disso, esta tela deverá chamar uma outra tela, portanto conecte a ação do botão "Dados de acesso" ao método `abreDadosAcesso:`, conforme a listagem abaixo:

```
// ConfiguracoesController.h
#import <UIKit/UIKit.h>

@interface ConfiguracoesController : UIViewController
- (IBAction)abreDadosAcesso:(id)sender;

@end
```

Para a tela de "Dados de acesso", crie um novo controlador chamado "*DadosAcessoController*" e adicione alguns campos, conforme a figura 4.10. Para esta tela não há necessidade de código adicional por enquanto.

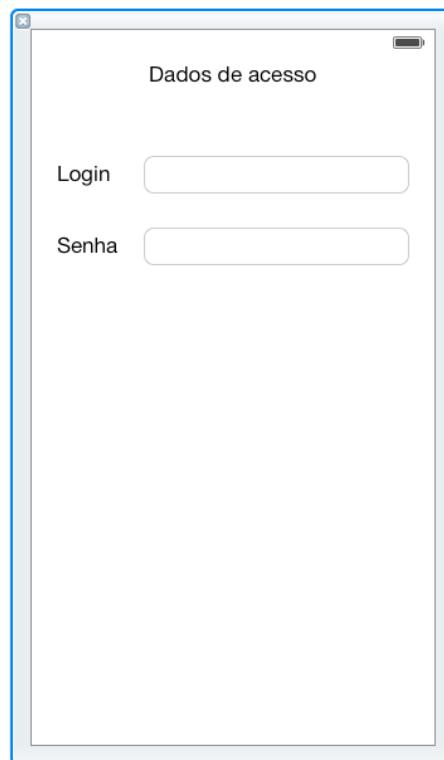



Figura 4.10: Exemplo das configurações de acesso

Agora, no arquivo `ConfiguracoesController.m`, implemente o corpo do método `abreDadosAcesso:` para que o `DadosAcssoController` seja aberto quando solicitado, conforme demonstrado abaixo:

```
- (IBAction)abreDadosAcesso:(id)sender {
    DadosAcessoController *c = [[DadosAcessoController alloc] init];
    [self.navigationController pushViewController:c animated:YES];
}
```

Execute o aplicativo (`Command + R`) e navegue entre os controladores, e veja como o `UINavigationController` cuida da parte de transição entre eles. Neste ponto deverá ser possível navegar conforme demonstrado na figura 4.11.

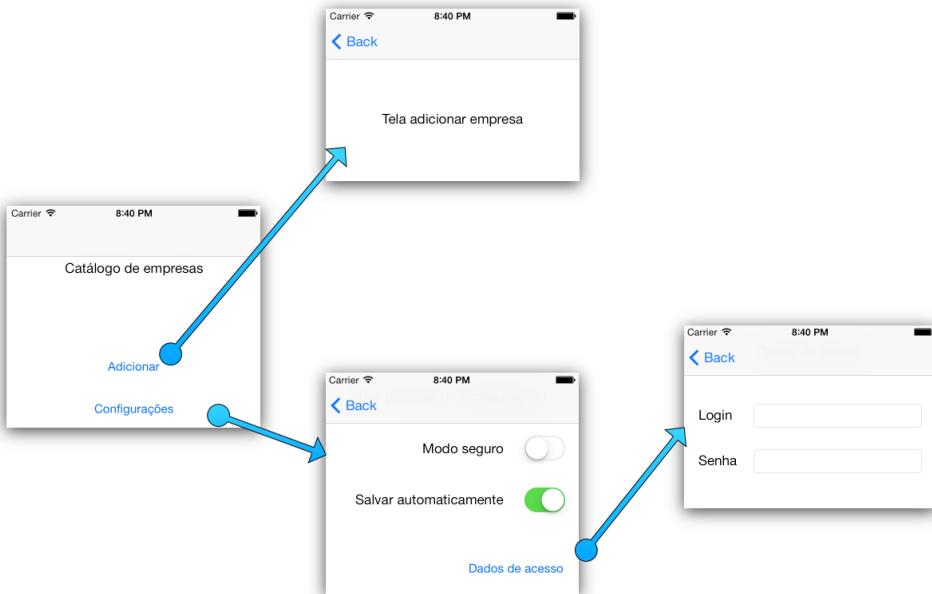



Figura 4.11: Estrutura da navegação

4.10 Esconder a barra superior de navegação

Certas vezes não há a necessidade de deixar a barra de navegação superior (a "Navigation Bar") do `UINavigationController` visível o tempo todo, caso ela não agregue funcionalidade, e o ideal seria escondê-la e mostrar somente quando necessário. Para obter este resultado basta executar o método `setNavigationBarHidden:animated` do `UINavigationController`, lembrando que ele é representado pela propriedade `navigationController`. Adici-

one a linha de código abaixo no método `viewWillAppear:animated` da classe `RootViewController`, e rode novamente o aplicativo.

```
// RootController.m
-(void) viewWillAppear:(BOOL)animated {
    [self.navigationController setNavigationBarHidden:YES animated:YES];
}
```

Contudo, quando navegamos na hierarquia para outro controlador, a barra de navegação continua escondida, impossibilitando de navegarmos para o nível anterior. Para solucionar este problema basta fazer o processo inverso nas classes `ConfguracoesController` e `AdicionarController`, conforme demonstrado abaixo:

```
// Nos controladores Configurações e Adicionar
-(void) viewWillAppear:(BOOL)animated {
    [self.navigationController setNavigationBarHidden:NO animated:YES];
}
```

Rode novamente o aplicativo (menu *Product* -> *Run*) e veja a diferença no funcionamento.

Capítulo 5

Storyboards

Historicamente, a maneira de construir aplicações iOS consistia em criar uma série de controladores, cada um de maneira independente do outro, e frequentemente com arquivos `.xib` para a criação do layout. Como vimos no capítulo 4, ainda é possível construir aplicações desta forma, porém agora existe a opção de utilizar storyboards no lugar de controladores independentes.

A grande vantagem de storyboards é que eles nos permitem definir visualmente como as interações entre as muitas telas dos aplicativos irão ocorrer, tudo isso em um único arquivo, o que nos dá uma visão ampla de todas as rotas possíveis. Aplicativos costumam ter um único storyboard, embora tecnicamente seja possível haver mais deles.

Tenha em mente que storyboards são uma maneira de criar controladores e definir como eles interagem entre si - por exemplo, como ir de um para o outro -, porém nada muito além disso. Tudo o que já foi mostrado no livro, e tudo mais o que ainda será demonstrado, pode trabalhar *em conjunto* com storyboards.

Os termos de storyboards

Quando falamos de storyboards em iOS, frequentemente veremos os seguintes abaixo. Como curiosidade, este é um termo que a Apple emprestou de outras artes, como filmes e teatro.

- **Scene**: termo em inglês para "*Cena*", daquela mesma de teatro ou TV, que é cada tela em si. No livro utilizaremos o termo em inglês, pois é algo bem próprio de storyboards e lhe ajudará em futuras pesquisas sobre o assunto. Na prática Scenes e controladores (vide abaixo) estão diretamente ligados, não sendo possível ter um sem o outro.
- **View Controller**: já vimos bastante sobre controladores, e eles formam uma parte fundamental de storyboards, trabalhando diretamente em conjunto com scenes. Tudo o que você já aprendeu sobre controladores no livro se aplica no caso de storyboards.
- **Segue**: é o *conceito* que defini a transição de uma scene para outra. O termo em inglês e português são iguais.

5.1 Projeto e conceitos gerais

Crie um novo projeto do tipo "Single View Application" com as configurações padrão, chamado "*DemonstracaoStoryboards*", e em seguida abra o arquivo " `Main.storyboard`". A imagem 5.1 mostra a estrutura inicial de uma storyboard.

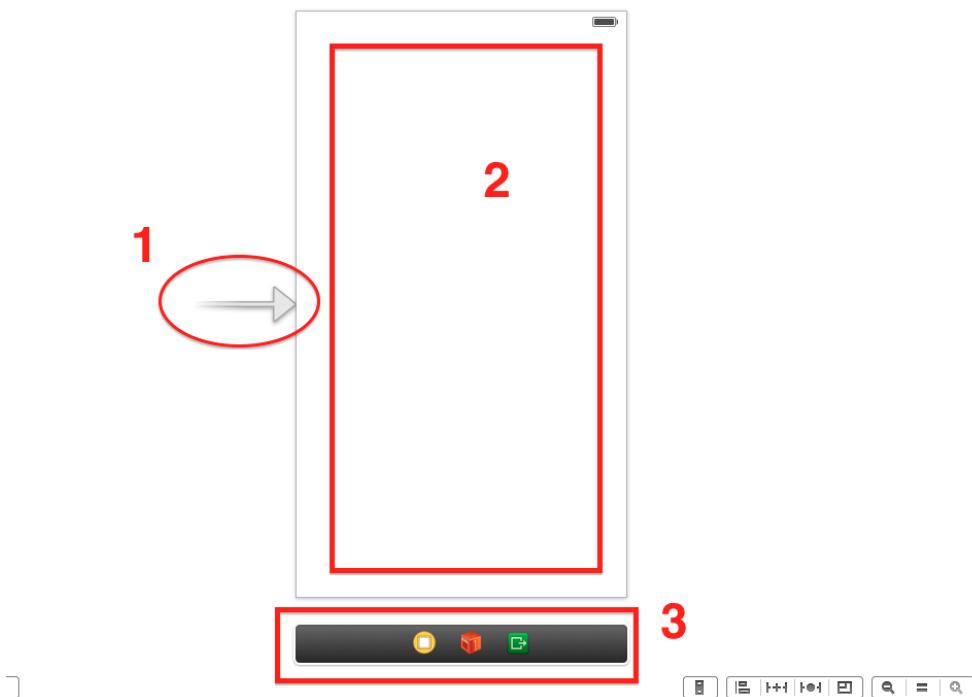



Figura 5.1: A estrutura inicial de uma storyboard

A seta indicada pelo item 1 informa que este controlador é o ponto de partida (controlador inicial) da storyboard, o item 2 é a Scene (que é a porção da view), enquanto que o item 3 é o controlador propriamente dito. Portanto, se você quiser definir propriedades da Scene (ou da view, utilizando um outro termo) clique em qualquer área da região branca, indicada pelo número 2. Por outro lado, caso queira definir propriedades do controlador, como título, opções de transição para outras Scenes, métricas de simulação de interface e afins, clique na barra preta localizada na parte inferior da Scene, indicada pelo número 3 na imagem - na prática, o controlador é o elemento amarelo localizado na extremidade esquerda da barra preta.

Neste capítulo será mostrado o funcionamento de storyboards sem focar em detalhes de um programa específico - mais adiante no livro construíremos um aplicativo mais completo que também fará uso mais extensivo de storyboards, ajudando assim a firmar os conceitos. Por enquanto, o importante é focar na maneira como ele funciona, o que é bastante simples. O aplicativo consiste em uma serie de telas fictícias, com alguns níveis de navegação auxiliados por um por navigation controller,

com os quais serão demonstrados as formas de conexão e navegação de um Segue para outro, como invocar manualmente um Segue, e como retornar para o início da app sem ter que voltar manualmente tela por tela.

A imagem 5.2 mostra o estado da storyboard do projeto desta capítulo quando estiver finalizada. A tela foi diminuida de tamanho para poder caber tudo em uma única imagem. Note que é possível visualizar toda a estrutura de navegação do aplicativo, sabendo quando é a tela inicial e os possíveis caminhos de uma tela para outra.

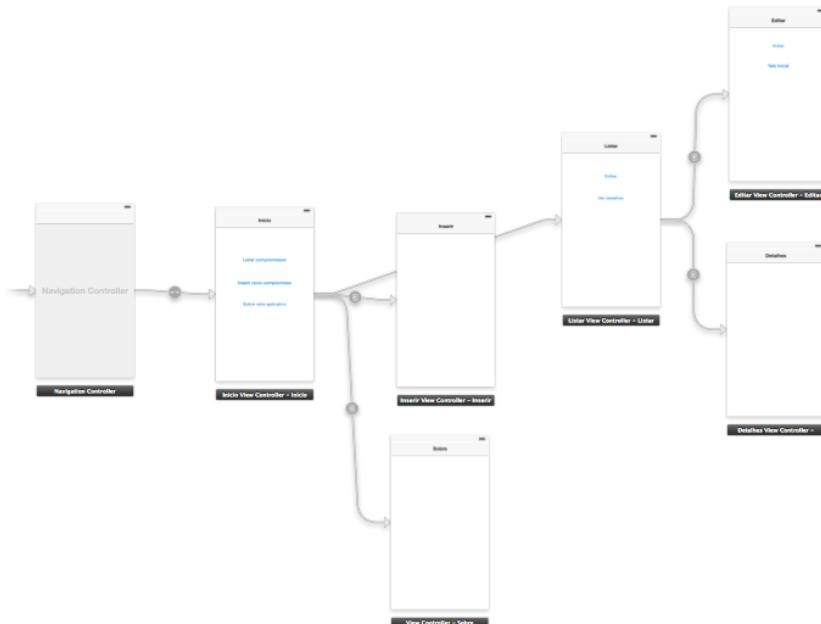



Figura 5.2: Resultado final da storyboard deste capítulo

Ao abrir o arquivo “ `Main.storyboard`” você encontrará uma tela como a da imagem 5.1, onde já há um controlador, o qual será a tela (Scene) inicial do aplicativo. Insira neste controlador 3 botões, chamados “*Listar compromissos*”, “*Inserir novo compromisso*” e “*Sobre este aplicativo*”, respectivamente. A imagem 5.3 mostra o resultado.

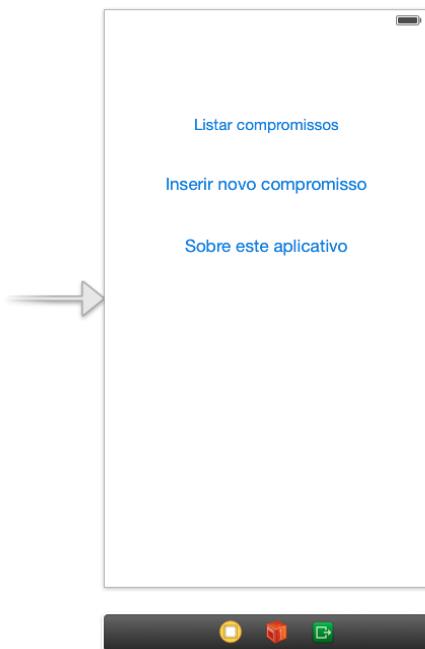



Figura 5.3: Tela inicial do aplicativo

Nível de zoom e organizar controladores na storyboard

Uma coisa ruim do editor de storyboards é que, quanto mais controladores você adicionar, mais difícil fica visualizá-los e organizá-los no editor, o que é ainda mais sofrível caso você tenha um monitor pequeno (ou, pior, apenas a tela do notebook, como muitas pessoas). O time da Apple ainda precisa trabalhar bastante em diversas questões de usabilidade.

Utilize os controles disponíveis no canto inferior direito da tela para alterar o nível de zoom do editor, conforme mostra a imagem [ref-labl sb2.png]. O botão do meio, representado pelo símbolo de igualdade ("=") retorna o zoom a 100%. Já para arrastar o controlador para outra parte da tela (como quando quiser mudá-lo de posição, por exemplo) clique e arraste a barra preta.

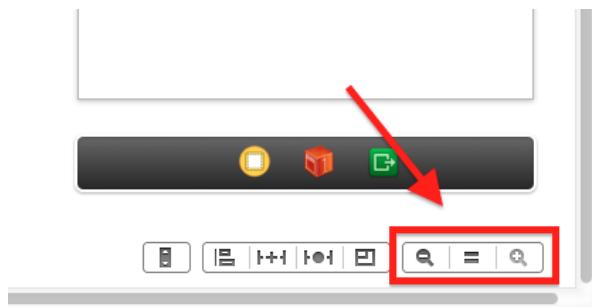



Figura 5.4: Controles de zoom

Construir a Segue 'Sobre'

Como você pode deduzir pela imagem 5.3, existem três Segues para as quais é possível navegar a partir da Segue inicial. Vamos começar pela "*Sobre este aplicativo*". Com o " `Main.storyboard`" aberto, abra a Object Library (`View -> Utilities -> Show Object Library`) e arraste para a storyboard um item do tipo "View Controller", e em seguida adicione neste controlador um `UILabel` com o texto "*Demonstração App 1.0*". A imagem 5.5 mostra o resultado parcial.

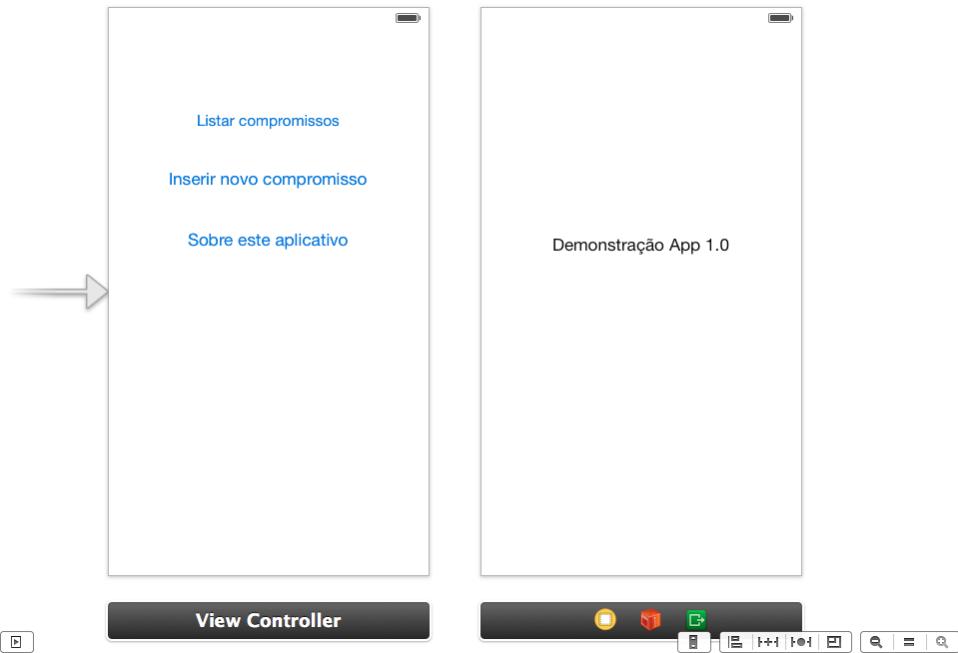



Figura 5.5: Resultado parcial com a tela 'Sobre'

Neste ponto temos dois controladores, porém sem qualquer ligação entre eles. O que precisa ser feito então é informar que, a partir do toque no botão "*Sobre este aplicativo*", o controlador "*Sobre*" seja adicionado. Anteriormente a storyboards este processo era feito manualmente, através da criação de uma `action` para o botão, e a manual instanciação do controlador. Já com storyboards a história fica bem mais fácil, podendo ser feito tudo visualmente pelo editor, sem qualquer necessidade de código.

Clique no botão "*Sobre este aplicativo*", segure `CTRL` e clique e arraste em direção ao controlador "*Sobre*", de tal forma que sua área fique azulada, e então solte o clique. Isso fará com que uma conexão - ou melhor, a Segue - seja feita do botão para este controlador, conforme mostra a imagem 5.6. Na tela popup que aparecer, selecione o item "*Push*".

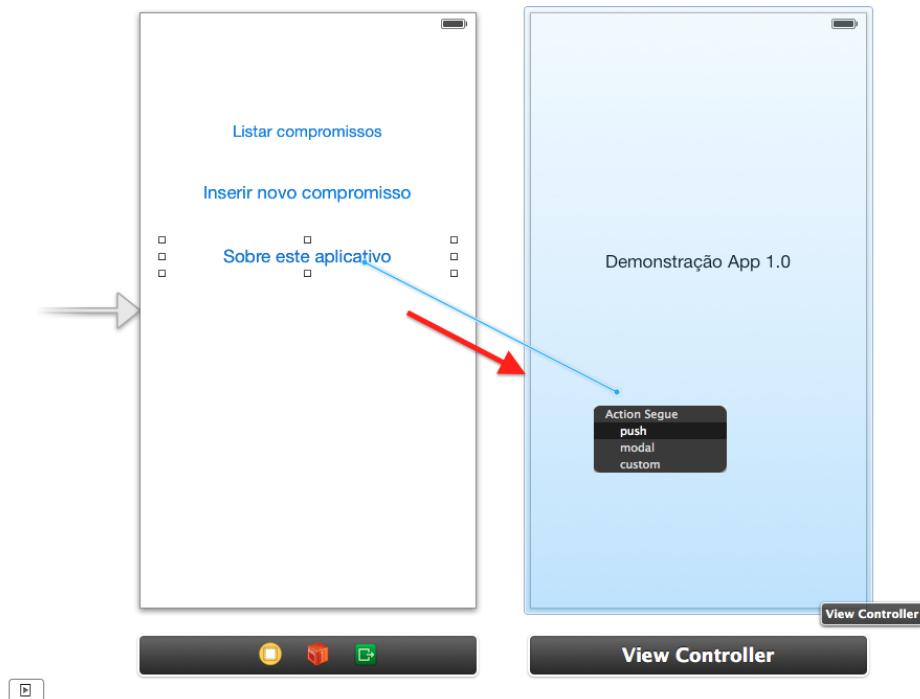



Figura 5.6: Criar Segue do botão para o controlador

Rode o aplicativo e clique no botão. O que aconteceu? Provavelmente o seguinte erro:

```
Terminating app due to uncaught exception 'NSGenericException', reason:
'Push segues can only be used when the source controller is managed by
an instance of UINavigationController.'
```

Esta mensagem de erro está dizendo que eventos Segue do tipo "*Push*", que foi o que selecionamos no popup na hora de fazer a conexão, precisam que exista um `UINavigationController` associado, caso contrário não é possível navegar entre as Scenes. Para resolver isso é bem simples: selecione o controlador que tem os botões, já que ele é a primeira tela do aplicativo, e então vá ao menu " `Editor -> Embed In -> Navigation Controller`", que o Xcode irá adicionar e configurar ele para nós. O resultado deverá ficar como o da imagem 5.7

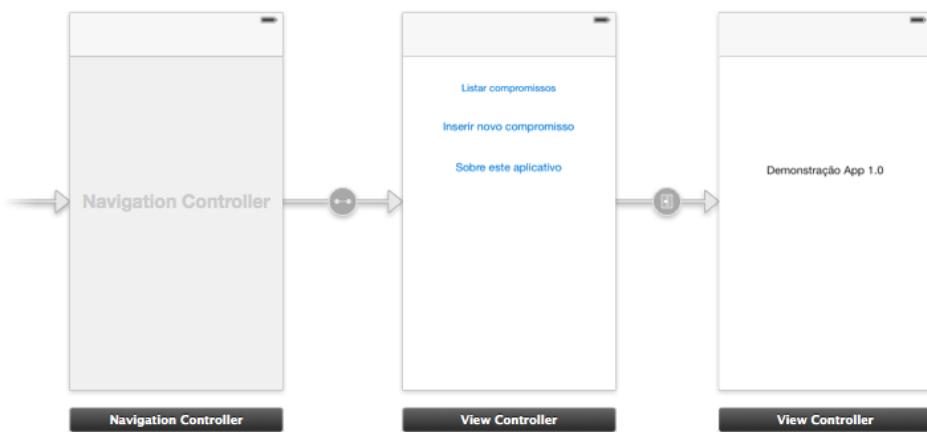



Figura 5.7: Storyboard após a adição do navigation controller

Como pode ver, o primeiro controlador da storyboard (representado pela seta mais a esquerda) passou a ser o navigation controller, que na prática é um controlador especial que gerencia a navegação entre outros controladores. Se você selecioná-lo (lembre-se que para selecionar o controlador é necessário clicar na barra preta) e abrir o Attributes Inspector (“ `View -> Utilities -> Show Attributes Inspector`”) verá que ele está marcado como sendo a Scene inicial, conforme mostra a imagem 5.8.

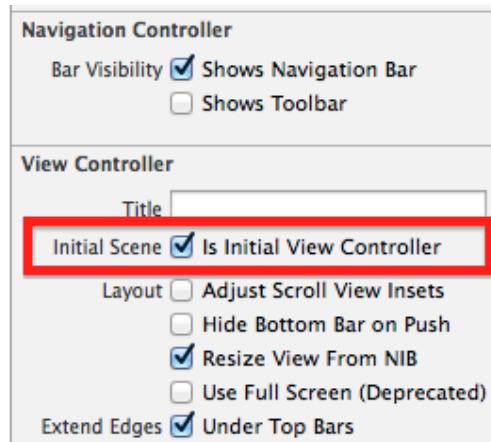



Figura 5.8: Navigation controller definido como sendo o primeiro

Rode o aplicativo novamente e clique no botão, e veja que agora ele funciona corretamente.

Ver a conexão e tipo de transição

Se você clicar na seta entre dois controladores, poderá visualizar o botão com o qual a ação está associada, conforme mostra a imagem 5.9, onde o botão é identificado com uma seleção azul. De quebra, no Attributes Inspector é possível modificar o tipo de transição, que no nosso caso foi definida como "Push".

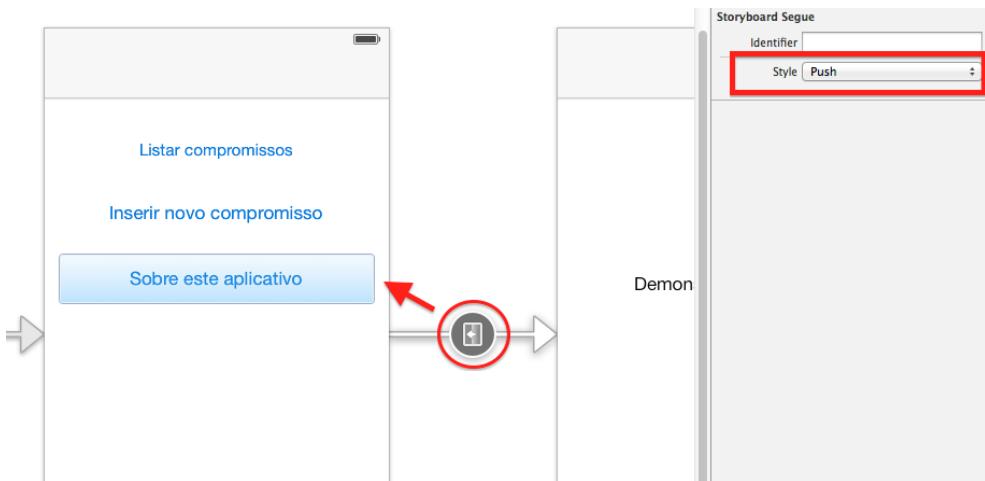



Figura 5.9: Verificar com qual botão a transição da Segue está associado

Título dos controladores

Como agora temos um navigation controller, ele por padrão adiciona uma barra superior nos controladores, onde é possível definir um título para a tela. Além disso, no canto superior esquerdo ele também coloca automaticamente um botão para voltar para a tela anterior, como você já deve ter notado. Seguindo a imagem 5.10, dê um duplo clique na região de título indicado pela seta, e escreva um texto curto e claro para identificar o controlador em questão - por exemplo, "Início" e "Sobre". Veja ainda que, na seção "*Navigation Item*" do Attributes Insepector, existem três campos de texto, sendo que os mais interessantes são "*Title*", para o texto que irá aparecer na barra (é o mesmo texto definid pelo duplo clique) e "*Back Button*", que é o texto que irá ser utilizado no botão para retornar a tela anterior - caso nenhum valor seja informado, o navigation controller irá automaticamente utilizar o título da tela anterior. Obs: o texto do botão "*Back Button*" deve ser definido no controlador "anterior", ou seja, aquele para qual a navegação irá retornar. Faça alguns testes para se acostumar com o funcionamento.

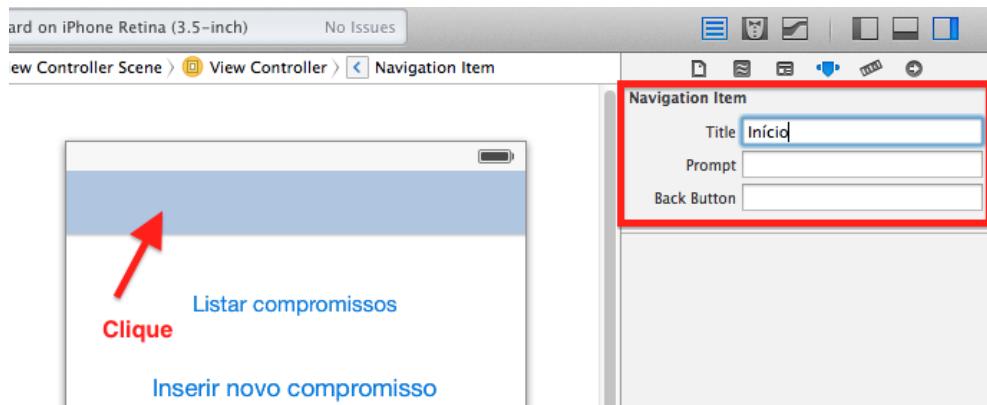



Figura 5.10: Propriedades de navegação

5.2 Adicionar os demais controladores

Agora que sabemos o funcionamento de storyboards, vamos adicionar os demais controladores. O processo é igual para todos, mudando apenas o título de cada controlador. Faça assim:

1) Adicione na storyboard um componente do tipo “ `View Controller`” localizado na Object Library (`View -> Utilities -> Show Object Library`)

2) Crie a Segue do botão “*Listar Compromissos*” para este novo controlador (clique no botão e depois `CTRL + clique e arraste`), escolhendo “*Push*” como ação. Obs: isso não pode ser feito se o zoom estiver habilitado, o Xcode não permite

3) Coloque um título na barra superior deste controlador (por exemplo, “Listar”)

4) Repita os passos 1 a 3 para a tela “*Inserir novo compromisso*”, com o título “Inserir”

Neste momento o seu resultado deverá estar como o da imagem 5.11. Rode o aplicativo e navegue para certificar-se que tudo funciona.

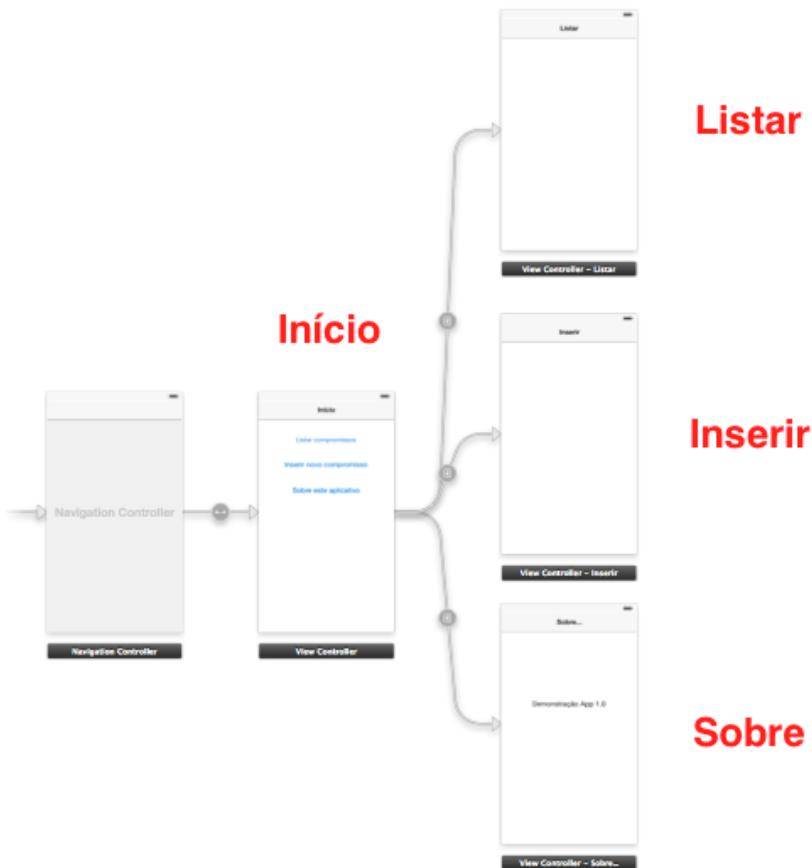



Figura 5.11: Propriedades de navegação

A tela de Listar e seus controladores, via código

A tela "Listar" permite duas ações adicionais: editar e ver os detalhes de um determinado registro. Abra o controlador com o título "Listar" e adicione dois novos botões, um com o texto "Editar" e outro com o texto "Ver Detalhes", **porém desta vez não crie as Segues** ainda, pois iremos fazer de uma maneira alternativa, via código. Aprender a fazer via código é útil quando for necessário navegar para outra Segue de uma maneira não tradicional, como a interação com a célula de uma Table View, ou um botão com ações condicionais (como um botão "Login" que depois vira

"Logout").

Existem duas coisas que precisam ser feitas: a primeira é que a Segue deve ser criada sem estar associada com um botão específico, mas sim "de controlador para controlador". Isso é feito criando a Segue através do item "View Controller" na barra presta, utilizando a mesma técnica de " `CTRL+clique e arraste`", conforme mostra a imagem 5.12.

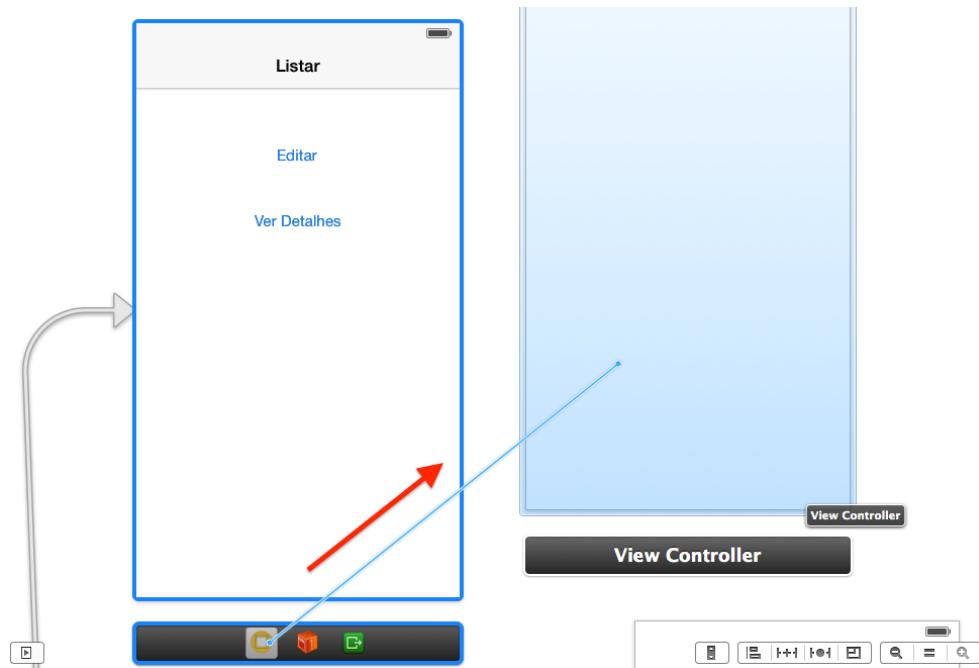



Figura 5.12: Criando a Segue sem associar ela a um botão

Repita o processo mais uma vez, agora para o outro controlador (lembre-se: um para editar, e outro para ver detalhes). O resultado desta parte deverá estar como a imagem 5.13.

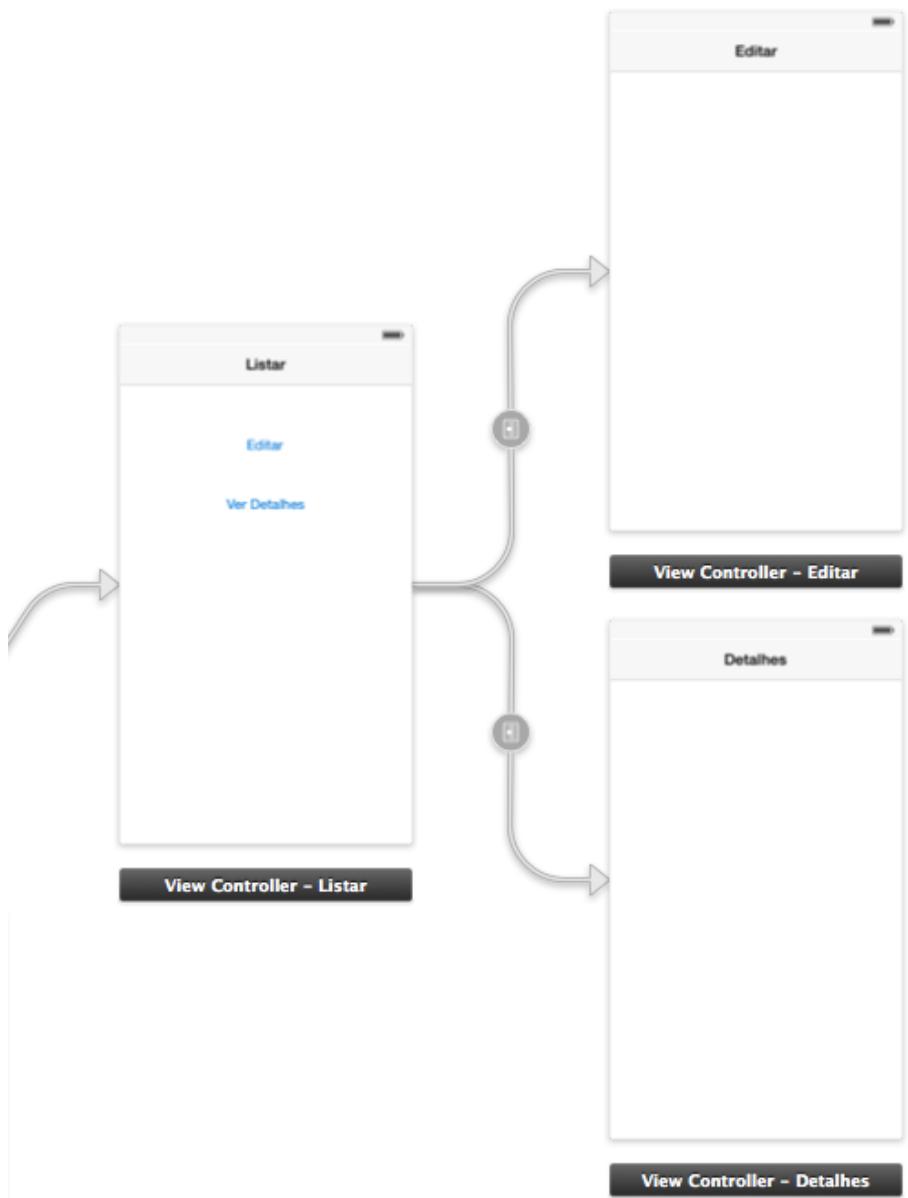



Figura 5.13: Segues criadas da tela de Listar para Editar e Detalhes, sem associação com algum botão

Para que seja possível navegar entre Segues via código é necessário adicionar identificadores *únicos* a elas. Para isso, clique na seta que une as Segues e preencha o campo "*Identifier*" no Attributes Inspector, conforme mostra a imagem 5.14. Lembre-se de utilizar nomes únicos para cada segue. Como sugestão, utilize uma combinação do nome da Segue de origem e de destino. No caso deste exemplo, "Listar -> Editar" deve ser chamada de "`listarParaEditarSegue`" e "Listar -> Detalhes" de "`listarParaDetalhesSegue`".

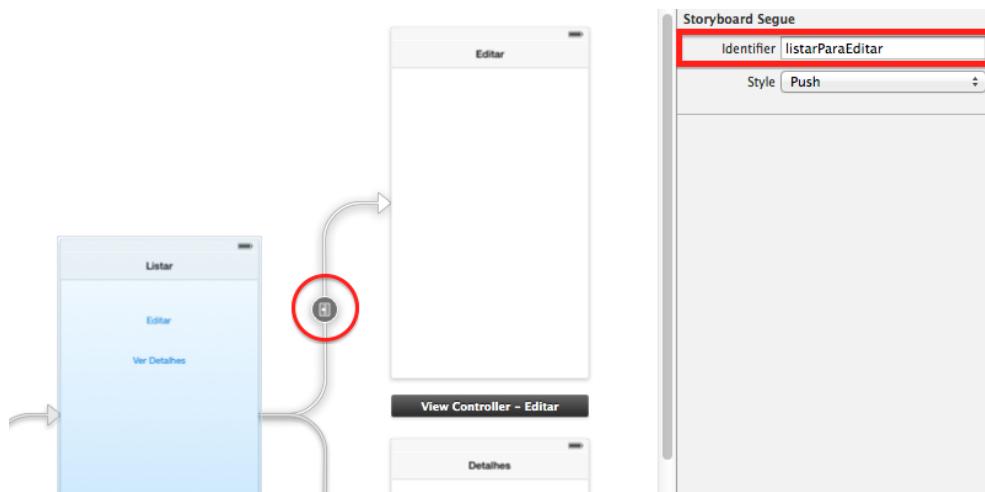



Figura 5.14: Adicionar identificar a uma Segue

Até agora todo o trabalho que realizamos foi unicamente na storyboard, sem precisar criar uma única classe ou linha de código. Contudo, sabemos que na vida real as regras de negócio são escritas em Objective-C, e para isso é necessário ter uma classe customizada associada ao controller, como já vimos diversas vezes no livro. Como agora queremos ir da Segue "Listar" para outras duas, vamos criar uma classe especial para ela.

Vá ao menu "`File -> New -> File...`", selecione "*Objective-C Class*" e clique em "*Next*". No campo "Class" insira o valor "ListarViewController", e no campo "Subclass of" certifique-se que a opção "UIViewController" esteja seleciona, pois queremos criar um controlador customizado. As opções "*Targeted for iPad*" e "*With XIB for User Interface*" devem estar *desmarcadas*.

Agora volte ao storyboard, selecione a Segue "Listar", abra o "Identity Inspector" ("`View -> Utilities -> Show Identity Inspector`") e na seção "*Cus-*

tom Class" insira o valor " `ListarViewController`", conforme mostra a imagem 5.15.

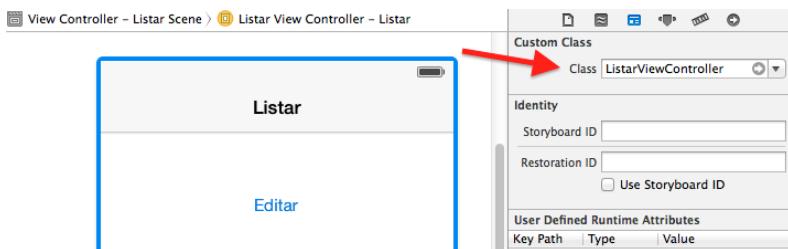



Figura 5.15: Identificador da Segue

Tendo feito isso, abra o Assistant Editor (" `CTRL+ENTER`") e crie as actions para os dois botões. Como sugestão, para o de Editar chame a action de " `abrirEditarScene`", e " `abrirDetalhesScene`" para o de Detalhes. O arquivo `ListarViewController.h` deverá estar como o código abaixo:

```
#import <UIKit/UIKit.h>

@interface ListarViewController : UIViewController

- (IBAction)abrirEditarScene:(id)sender;
- (IBAction)abrirDetalhesScene:(id)sender;

@end
```

Todo controlador já contêm um método chamado `performSegueWithIdentifier`, que serve para navegar dele para outra Segue utilizando o identificador definido anteriormente. O arquivo `ListarViewController.m` deve conter a implementação dos métodos "abrir" conforme abaixo:

```
- (IBAction)abrirEditarScene:(id)sender {
    [self performSegueWithIdentifier:@"listarParaEditarSegue"
          sender:sender];
}

- (IBAction)abrirDetalhesScene:(id)sender {
    [self performSegueWithIdentifier:@"listarParaDetalhesSegue"
```

```
            sender:sender];
}
```

Rode o aplicativo e interaja com os botões para ver o resultado.

5.3 Navegar de volta diretamente para a Segue inicial

Uma das coisas atreladas ao uso de Navigation Controllers é que, a medida em que navegamos de um controlador para outro, eles vão sendo colocados em uma pilha, um em cima do outro, e para voltar para as telas anterior é preciso ir clicando no botão "Voltar" (aquele do canto superior esquerdo). Só que quando queremos voltar para algum mais distante, é necessário fazer isso várias vezes.

Com storyboards existe uma maneira de tornar este processo um pouco mais fácil, através do uso da ação de "Saída", do termo em inglês "*Exit*". Na verdade, o termo utilizado pela documentação é "*Unwind Segue*", porém no Xcode a palavra "*Exit*" é utilizado, para nos confundir um pouco!

Funciona assim, em dois passos: adicionamos um método com qualquer nome na Segue para onde deseja-se *retornar* (por exemplo, " `resetarNavegacao`") e que receba um argumento do tipo `UIStoryboardSegue`, e ai na Segue de *origem*, que irá disparar a ação, fazemos um link entre o botão e a ação de *Unwind*, ou *Exit*.

Para demonstrar, vamos definir que queremos voltar da Segue "Detalhes" (`Inicio -> Listar -> Detalhes`) para o início da app. Para isso, adicione uma nova classe chamada " `InicioViewController`" que herde de `UIViewController` como feito anteriormente (não esqueça de informar o nome dela no campo "*Custom Class*" do Identity Inspector na Storyboard). Ai abra o arquivo " `InicioViewController.m`" e adicione o seguinte código:

```
-(IBAction)resetarNavegacao:(UIStoryboardSegue *) segue {
    // Nao precisa de código
}
```

Veja que o corpo do método é vazio mesmo, embora você possa colocar algum código caso queira - porém isso não é obrigatório.

Para o segundo passo, vá até a Segue "Editar" na Storyboard (não é necessário criar uma classe customizada) e adicione um botão com o texto "Voltar para o

início". Em seguida, selecione o botão, segure `CTRL` e clique + arraste para o botão verde, conforme mostra a imagem 5.16. No popup que abrir, selecione o item " `resetarNavegacao`".

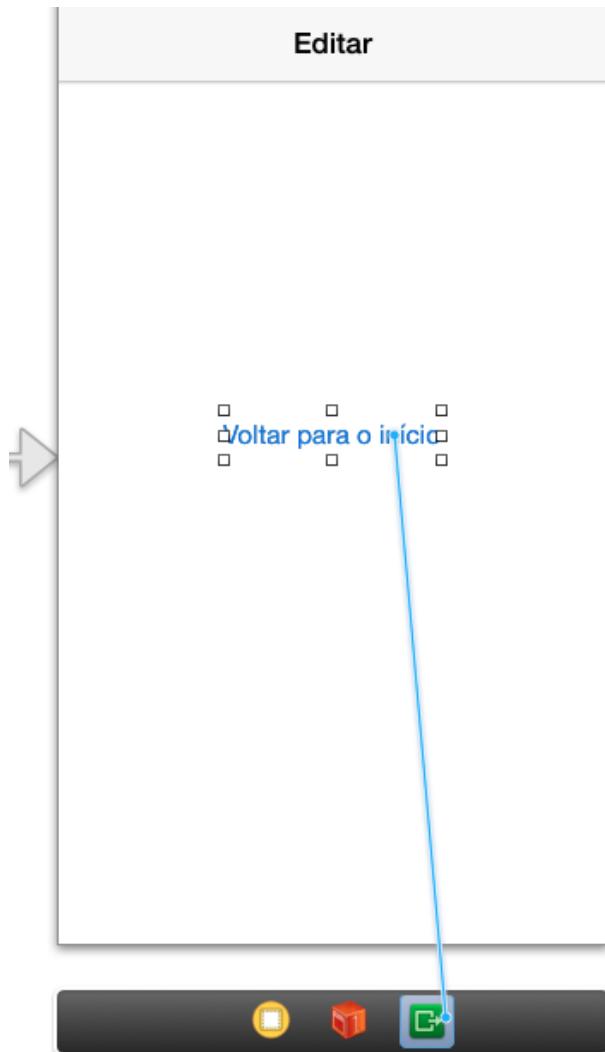



Figura 5.16: Associar a ação Unwind a um botão

Rode o aplicativo e veja o resultado.

Informar se a Segue pode aceitar a ação de Unwind

Existe um método de implementação opcional que serve para indicar se aquele controlador pode ou deve aceitar a ação de Unwind, com base em condicionais criadas pelo desenvolvedor, como local (Segue) de origem e de destino. Caso este método retorne " YES". Um outro caso que ele pode ser necessário é quando a ação de Unwind passa por diversas Segues intermediárias e o retorno acaba parando "no meio do caminho", mesmo quando não deveria. Nestes casos, portando, implemente o método e retorne " NO". Veja o código abaixo:

```
-(BOOL) canPerformUnwindSegueAction:(SEL)action
    fromViewController:(UIViewController *)fromViewController
    withSender:(id)sender {

    return YES;
}
```

Retornar " YES" indica que o controlador pode manipular a ação de Unwind, onde neste caso deverá ter também implementado o método indicado no popup do botão verde, como feito anteriormente. Já " NO" faz a ação passar direto.

5.4 Passar dados de uma Segue para outra

Um detalhe fundamental com o uso de storyboards é que os controladores são criados automaticamente, portanto não temos como passar objetos e outras referências pelo construtor, como geralmente é feito da forma manual. A abordagem consiste, portanto, em sobrescrever um método especial existente em `UIViewController` chamado " `prepareForSegue`", o qual é executado momentos antes da transição para a próxima Segue.

Este método passa como argumento informações sobre a Segue que está para ser executada, incluindo seu identificador e a referência para o controlador. Com isso, objetos podem ser atribuídos através de propriedades. Para demonstrar o conceito é necessário mexer em dois controladores - o de origem e o de destino. No caso, a intenção é passar dados para o controlador "Detalhes" (que ainda não foi criado), vindo do "Listar" (o qual já criamos).

Primeiro crie um novo controlador chamado " `DetalhesViewController`", e associe-o à tela de detalhes através do campo "*Custom Class*" no Identity Inspector. Além disso, adicione também um `UILabel`, fazendo a devida ligação com o ar-

quivo " `DetalhesViewController.h`", através da técnica de " `CTRL + clique e arraste`". A imagem 5.17 serve de referência.

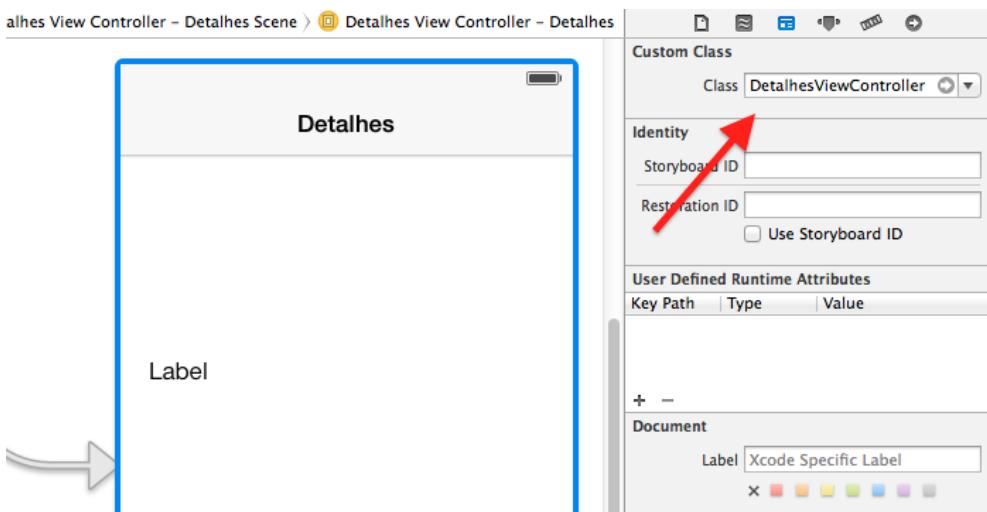



Figura 5.17: Segue de Detalhes com a classe definida

Aproveitando que estamos mexendo no controlador de Detalhes, adicione uma propriedade chamada " `descricaoDetalhes`", que será aquela preenchida antes de navegar para a Segue, através da tela de listagem. O código deverá ficar assim:

```
@interface DetalhesViewController : UIViewController

@property (weak, nonatomic) IBOutlet UILabel *infoLabel;
@property (nonatomic, retain) NSString *descricaoDetalhes;

@end
```

Feito isso, o próximo passo é interceptar o evento de navegação para a Segue, e definir um valor para a propriedade " `descricaoDetalhes". Abra a classe "`ListarViewController.m

```
// Obs: não esqueça de importar o arquivo "%%DetalhesViewController.h%%"
-(void) prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if ([segue.identifier isEqualToString:@"listarParaDetalhesSegue"]) {
        DetalhesViewController *dc = segue.destinationViewController;
```

```
        dc.descricaoDetalhes = @"Conteúdo vindo da tela de listagem";
    }
}
```

O código não tem maiores mistérios: primeiro verifica-se se a Segue que irá ser exibida é a que queremos manipular, através do identificador definido anteriormente, e em seguida simplesmente pegamos a referência ao controlador para definir algum valor a propriedade “ `descricaoDetalhes`”. Esse padrão pode - aliás, deve - ser repetido sempre que você quiser passar dados de uma Segue para outra.

Rode o programa e veja o resultado. Apareceu na tela o texto que foi definido? Provavelmente não, pois esquecemos de um detalhe: configurar o texto da `UILabel`. Abra o arquivo “ `DetalhesViewController.m`” e implemente o método “ `viewDidLoad`” conforme abaixo:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    _infoLabel.text = _descricaoDetalhes;
}
```

Simples, não?! Agora você já sabe tudo o que é necessário para trabalhar com storyboards.

Capítulo 6

Realizando operações com a Internet

Agora você já conhece todo o essencial da API do iOS. Além disso, sempre precisamos utilizar diversas bibliotecas extras, sejam elas da própria API fornecida pela Apple ou através de bibliotecas open source que realizam as mais diferentes tarefas.

Vamos aprender a usar uma biblioteca externa com uma situação bem comum: baixar arquivos pela rede para poder utilizá-los, sejam figuras, arquivos de configuração ou dados.

O código-fonte deste capítulo está disponível na pasta "VisualizadorImagens" e "ExemploDownload" (lembrando que o endereço do site com os códigos está na introdução do livro).

6.1 Conheça a biblioteca AFNetworking

Realizar operações de rede, como download ou upload de arquivos, além de interagir com APIs de serviços como Twitter e Facebook, são operações bastante comuns em aplicativos iOS. Com o amadurecimento da plataforma, diversas bibliotecas que antes existiam apenas para Mac OS foram portadas para iOS, tornando o acesso à documentação, exemplos e solução de problemas algo muito mais fácil de conseguir atualmente.

A biblioteca *AFNetworking*, que abordaremos neste capítulo, é uma das que mais se destacam na comunidade de desenvolvedores devido à sua API moderna e ativa comunidade. AFNetworking funciona tanto em iOS quanto em Mac OS X. Ela é feita sob a fundação de networking do próprio sistema operacional, tirando proveito do que há de melhor para trabalhar com comunicação de rede, tendo por padrão inclusive classes para lidar com conteúdo XML e JSON (além das operações normais), o que é de grande utilidade. A página oficial do projeto é https://github.com/AFNetworking/AFNetworking, e no livro utilizamos a versão 2.0.

O primeiro exemplo consiste em fazer o download de um arquivo para uma pasta dentro do nosso aplicativo, mostrando ao usuário uma barra de progresso à medida que o download é realizado.

Crie um novo projeto no Xcode do tipo *Single View Application* chamado "ExemploDownload".

Em seguida, baixe o projeto disponível no endereço https://github.com/AFNetworking/AFNetworking/zipball/master e descompacte-o em qualquer diretório. Neste pacote, copie a pasta "AFNetworking" para o diretório do projeto "ExemploDownload" onde se encontram as outras classes do projeto (AppDelegate, ViewController e outras).

Tendo feito isso, precisamos adicionar as classes do AFNetworking no nosso projeto. Para tanto, selecione o menu `File -> Add Files to "Exemplo Download"` e selecione o diretório do AFNetworking que você copiou no passo anterior (selecione o diretório mesmo, não os arquivos dentro dele). O resultado no Xcode deverá ficar como o da figura 6.1.

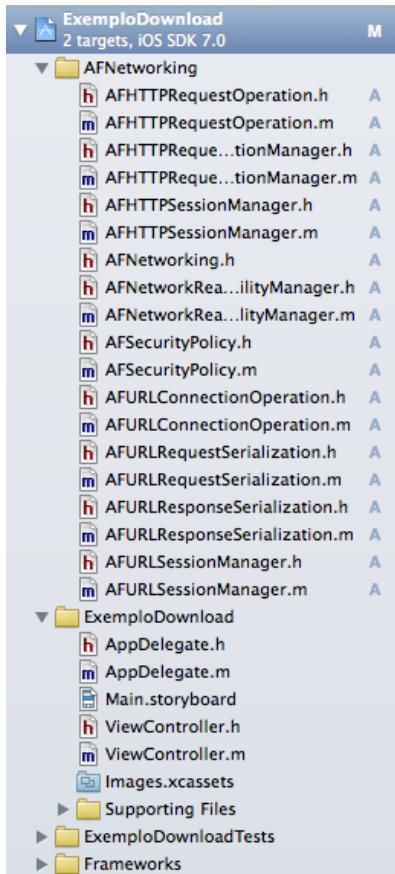
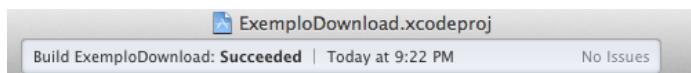



Figura 6.1: Estrutura do projeto após importar o AFNetworking

Compile o projeto (`COMMAND+B`), e se correu tudo sem problemas, você deverá ter um resultado como o da figura 6.2.

6.2 Criando a interface de download

O objetivo deste aplicativo é solicitar ao usuário o endereço de um arquivo a ser baixado, e mostrar o progresso a medida em que o download ocorre. Não faz diferença

se for um arquivo ZIP, Imagem ou qualquer outro formato, o ideal é apenas que seja grande o suficiente para ver a barra de progresso em funcionamento (ex: 3 MB para uma conexão lenta, 30 MB para conexões um pouco mais rápidas). Ao término do download será mostrado um alerta com o tamanho total do arquivo baixado.

Abra o arquivo *Main.storyboard* e adicione os seguintes componentes:

1) Label, com o texto "URL do arquivo"
2) Text Field, onde será informado o arquivo a baixar
3) Um Button com o texto "Download"
4) Um componente "Progress View", que será a nossa barra de progresso
5) Um componente "Activity Indicator View", que é aquela "rodinha" que fica girando enquanto alguma ação está acontecendo

A figura 6.2 mostra como deverá ficar a interface.

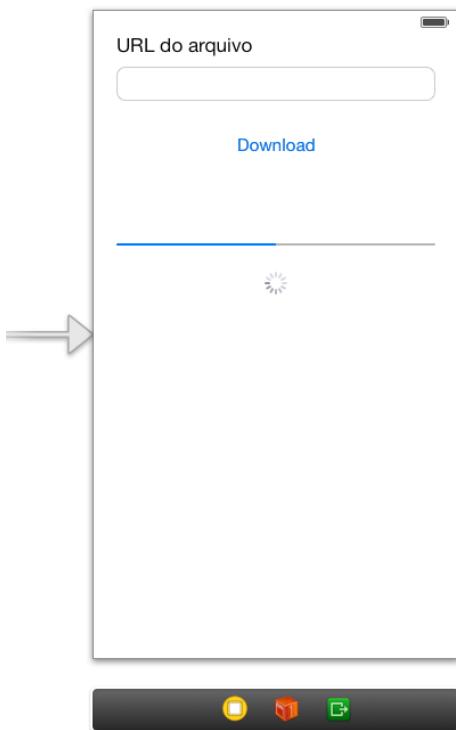



Figura 6.2: Resultado da construção da interface do aplicativo

6.3 Conectando os componentes com o código

Agora que temos a UI ("User Interface", ou interface com o usuário) criada, precisamos conectar os componentes ao código, a fim de podermos manipulá-los. Lembre-se que o arquivo `.storyboard` contém apenas a representação visual dos elementos de uma determinada tela, e a conexão com o código-fonte deve ser feita manualmente. Existem duas formas distintas de fazer isso:

1) Usando o "mini-wizard" do Xcode, que faz todo o trabalho pesado (já utilizada em capítulos anteriores)

2) Da forma tradicional (como era até o Xcode 3)

Dependendo do tamanho do seu monitor, a quantidade de painéis abertos no Xcode pode tornar a tela bastante "poluída", portanto pessoalmente costumo fazer o seguinte:

- Abra o Assitant Editor (*View -> Assistant Editor -> Show Assistant Editor*)
- Feche o painel Navigator (*View -> Navigators -> Hide Navigator*)
- Abra o painel Utilities (*View -> Utilities -> Show Utilities*)

Lembrando que esta é apenas uma configuração sugerida, porém você deve trabalhar da maneira que achar mais produtivo e confortável. Contudo, para as explicações abaixo assume-se que a configuração de telas sugerida é a que estará sendo utilizada.

A primeira maneira já vimos como fazer nos capítulos anteriores, que é aquela onde seguramos `CTRL` e arrastamos uma linha do componente até o arquivo `.h`. A segunda maneira consiste em primeiro declarar manualmente o código relevante no arquivo `.h`, para em seguida realizar as conexões.

Com os arquivos `Main.storyboard` e `ViewController.h` abertos lado a lado (veja a imagem 6.3), comece editando o arquivo `ViewController.h` para que fique conforme a listagem abaixo:

```
1 @interface ViewController : UIViewController
2 @property (nonatomic, weak) IBOutlet UITextField *downloadField;
3 -(IBAction)startDownload:(id)sender;
4 @end
```

Na linha 2 declaramos o campo de texto que será usado para informar a URL do arquivo a baixar, e na linha 5 é declarada a assinatura do método para iniciar o download em si.

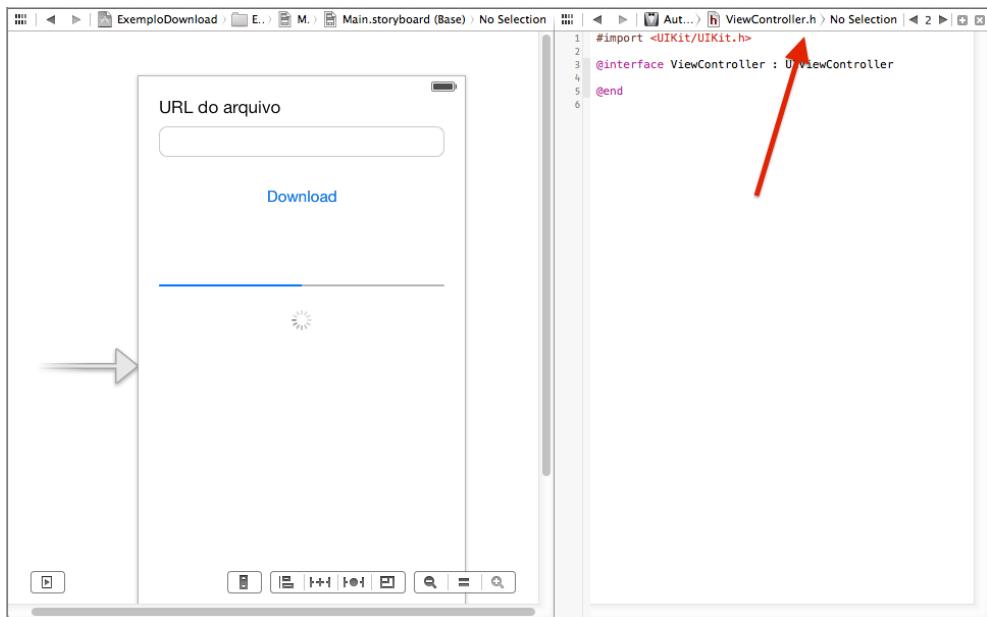



Figura 6.3: Main.storyboard e ViewController.h lado a lado

Agora faça os seguintes passos:

- Selecione o text field no arquivo Main.storyboard
- Abra o painel Connections, através do menu `View -> Utilities -> Show Connections Inspector`
- No painel Connections, clique e arraste a bolinha "*New Referencing Outlet*" para cima da declaração do UITextField no arquivo `.h`

Veja a figura 6.4 para referência.

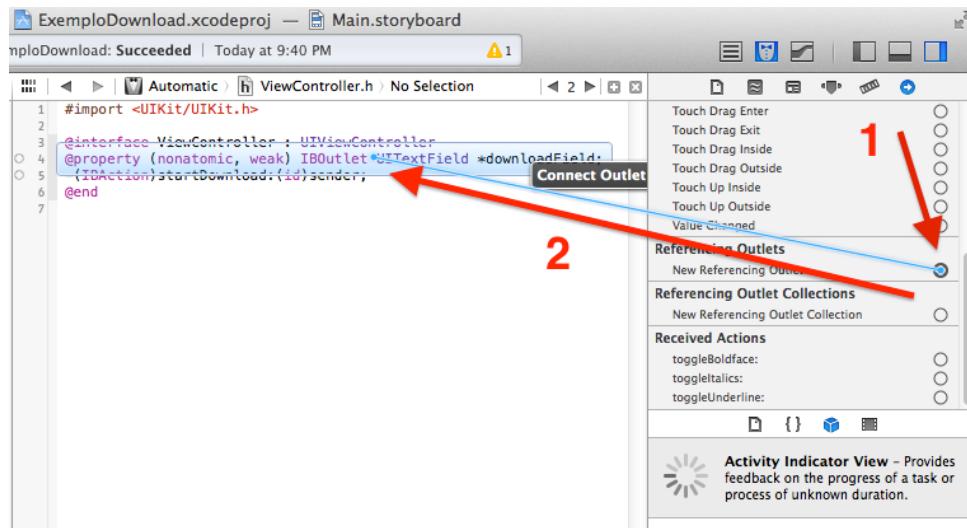



Figura 6.4: Passos para conectar o Outlet ao text field

Dessa forma, conectamos a propriedade `downloadField` que foi declarada no arquivo `.h` com o componente visual, permitindo manipulá-lo da maneira que for necessária. Caso contrário, o text field seria apenas um artefato puramente visual, sem qualquer utilidade.

O próximo passo é conectar a ação do botão que realiza o download. Os procedimentos são muito parecidos com o de conectar o text field, com a diferença que usaremos a `IBAction`. Os passos são:

- Selecione o botão no arquivo Main.storyboard
- No painel Connections aparecerá um grupo chamado "Send Events", com uma série de eventos possíveis que este botão pode receber. O que nos interessa é o *Touch Up Inside*, que é o responsável por lidar com um toque (ou "clique", se fosse um aplicativo web ou desktop). Arraste o conector do Touch Up Inside para *cima* da declaração da *IBAction* criada anteriormente. Se você fizer certo, irá aparecer um box dizendo "Connect Action"

Veja a figura 6.5 para referência.

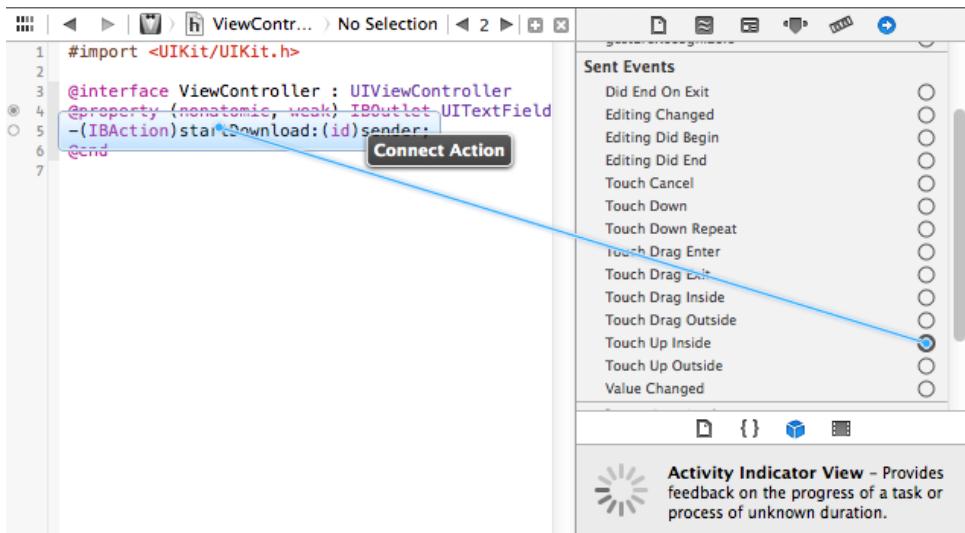



Figura 6.5: Conectando a ação do botão de download

Repita o mesmo procedimento para os componentes restantes: a barra de progresso (`UIProgressView`) e o indicador de atividade (`UIActivityIndicatorView`). Para a primeira, utilize o nome `progressBar`, e para o segundo, `loading`.

Após tudo isso, o código do arquivo `ViewController.h` deverá estar como o exemplo abaixo:

```
@interface ViewController : UIViewController
@property (nonatomic, weak) IBOutlet UITextField *downloadField;
@property (nonatomic, weak) IBOutlet UIProgressView *progressBar;
@property (nonatomic, weak) IBOutlet UIActivityIndicatorView *loading;
-(IBAction)startDownload:(id)sender;
@end
```

Volte para o "*Standard editor*" (`View -> Standard Editor -> Show Standard Editor`) e abra o arquivo `ViewController.m` (COMMAND+SHIFT+O) para implementar o corpo do método do botão de download. Isso é necessário porque criamos a conexão dele manualmente anteriormente. Veja no arquivo .m que o método `-(IBAction)startDownload:(id)sender` não existe no arquivo. Em qualquer lugar, implemente-o desta maneira:

```
-(IBAction)startDownload:(id)sender {

}
```

6.4 Realizar a operação de download

A implementação completa do método `startDownload:` é mostrada abaixo, e é um pouco mais longa do que os outros códigos que vimos até agora. Contudo, apesar do tamanho, são poucos os pontos que merecem uma explicação mais detalhada.

```
1  -(IBAction)startDownload:(id)sender {
2          NSURL *url = [NSURL URLWithString:_downloadField.text];
3      NSURLRequest *request = [NSURLRequest requestWithURL:url];
4      NSString *saveFilename =
5      [self downloadSavePathFor:url.lastPathComponent];
6
7      NSLog(@"Salvando o arquivo em %@", saveFilename);
8
9      AFHTTPRequestOperation *operation = [[AFHTTPRequestOperation alloc]
10         initWithRequest:request];
11
12     operation.outputStream = [NSOutputStream
13         outputStreamToFileAtPath:saveFilename append:NO];
14
15     [operation setCompletionBlockWithSuccess:
16          ^(AFHTTPRequestOperation *op, NSHTTPURLResponse *response) {
17              [_loading stopAnimating];
18              _loading.hidden = YES;
19              [self showMessage:@"Download finalizado com sucesso"];
20          }
21         failure:^(AFHTTPRequestOperation *op, NSError *error) {
22             [self showMessage:
23                 [NSString stringWithFormat:@"Erro no download: %@",
24                 [error localizedDescription]]];
25         }
26     ];
27
28     [operation setDownloadProgressBlock:
29         ^(NSUInteger read, long long totalRead,
30                                long long totalExpected) {
31         _progressBar.progress = (float)totalRead / (float)totalExpected;
```

```
32     }];
33 
34     _progressBar.hidden = NO;
35     _progressBar.progress = 0;
36     _loading.hidden = NO;
37     [_loading startAnimating];
38 
39     [operation start];
40 }
```

Muitas requisições com o AFNetworking, sejam para enviar ou receber dados, utilizam um *NSURLRequest* para fazer as operações. Esse código é construído nas linhas 2 e 3, enquanto que na linha 9 iniciamos o objeto que irá lidar com as requisições em si, através da classe *AFHTTPRequestOperation*. O objeto *NSURLRequest* encapsula tudo a respeito de uma requisição HTTP, incluindo a URL, os métodos (GET, POST, PUT, DELETE), os headers e corpo da requisição.

O código que se inicia na linha 15, `setCompletionBlockWithSuccess:`, utiliza um recurso avançado de Objective-C chamado *blocks*, que — falando de uma maneira bem geral — são objetos que contêm um comportamento específico, que será executado em um determinado momento. Blocks são tratados em detalhes no capítulo sobre Objective-C. O código completo do método inicia na linha 15, com a operação que deverá ser executada no caso de sucesso da operação, e termina na linha 26. A linha 21 contêm a declaração do block de código que será executado no caso da requisição falhar por algum motivo. A assinatura completa do método é `setCompletionBlockWithSuccess:failure:`. No caso da operação completar com sucesso, paramos o indicador de atividade e o escondemos, conforme as linhas 17 e 18.

Na linha 28 definimos a operação que será executada (que também utiliza *blocks*) toda vez que recebermos dados do servidor — no caso, pedaços do arquivo que iremos baixar. Na linha 30 atualizamos a barra de progresso com a porcentagem até o momento, lembrando que os valores da `UIProgressBar` vão de 0 até 1 (ou seja, 73% é representado como 0.73).

Os códigos definidos em blocos são sempre executados em um outro momento, ao contrário de métodos "normais". Eles são os chamados *callback methods*. O AFNetworking trabalha automaticamente em background, o que evita que a UI fique "congelada", o que faria o usuário pensar que o aplicativo travou.

O código do método `downloadSavePathFor:` está implementado abaixo:

```
1 -(NSString *) downloadSavePathFor:(NSString *) filename {
2     NSArray *paths =
3         NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
4         NSUserDomainMask, YES);
5     NSString *documentsPath = [paths objectAtIndex:0];
6     return [documentsPath stringByAppendingPathComponent:filename];
7 }
```

Na linha 1 e 2 pegamos a referência para o caminho do diretório *Documents* do aplicativo, e na linha 3 concatenamos com o nome do arquivo que será feito o download. Note que é um código relativamente complicado para uma tarefa tão simples; como você necessitará do caminho para o diretório *Documents* diversas vezes em seu aplicativo caso queira salvar arquivos, é uma boa prática implementá-lo em algum lugar que possa ser reaproveitado.

Por sua vez, o código do método `showMessage:` não tem nenhum mistério, como visto a seguir:

```
-(void) showMessage:(NSString *) message {
    UIAlertView *alert = [[[UIAlertView alloc] initWithTitle:@"Aviso"
        message:message
        delegate:nil
        cancelButtonTitle:@"OK"
        otherButtonTitles:nil] autorelease];

    [alert show];
}
```

6.5 Trabalhando com JSON e imagens remotas

Além da classe de uso geral `AFHTTPRequestOperation`, o AFNetworking vem com algumas outras classes bastante práticas para lidar com JSON, XML e carregamento assíncrono de imagens — esta última se destaca por ser uma grande mão na roda, pois nos livra de ter que programar muita coisa caso fossemos ter que fazer o processo manualmente.

JSON

JSON é um formato simples para transferir dados. É um subconjunto da notação de objeto de JavaScript, mas seu uso não requer JavaScript exclusivamente. É uma alternativa ao XML e também possibilita organizar os dados de forma hierárquica, conforme exemplificado abaixo:

```
// Exemplo de JSON, retirado da API do site 500px.com
{
    id: 54007966,
    name: "winner and loser",
    times_viewed: 2548,
    rating: 74.8,
    votes_count: 985,
    image_url: "http://.....",
    images: [
        {
            size: 4,
            url: "http://....."
        }
    ]
}
```

O próximo exemplo mesclará o uso de JSON com carregamento remoto e assíncrono de imagens, para criar um visualizador de imagens da comunidade fotográfica 500px (http://500px.com) . Ao iniciar, o aplicativo buscará no 500px a relação das últimas imagens enviadas pelos usuários, processar o resultado e mostrar as imagens em componente de scroll à medida que o usuário vai interagindo com o aplicativo. O código-fonte completo deste exemplo está localizado na pasta *VisualizadorImagens*, e na figura 6.6 você vê uma prévia do aplicativo final rodando.

Figura 6.6: App mostrando imagens públicas do 500px.com

Comece criando um novo aplicativo do tipo "*Single View Application*", com as mesmas configurações utilizadas no exemplo anterior e chame-o de "VisualizadorImagens". Depois de criar o projeto, não esqueça de incluir o pacote do AFNetworking, copiando o diretório como na app anterior.

A interface do aplicativo é bastante simples, consistindo apenas de um componente `UIScrollView`. Abra o arquivo `Main.storyboard` e mude a cor de fundo a view para um tom escuro, como RGB 40, 40, 40. Para isso, abra o *Attributes Inspector* (`View -> Utilities -> Show Attributes Inspector`) e clique no componente "Background", que abrirá a mini janela "Colors" (obs: caso não apareça nada para você, clique na área branca da tela do aplicativo que aparece no storyboard). Nela, selecione a opção "Color Sliders" (segundo da esquerda para a direita) e depois "RGB Sliders", no combo logo abaixo da lupa. Insira os valores e feche a janela Colors. Veja a figura 6.7 para referência.

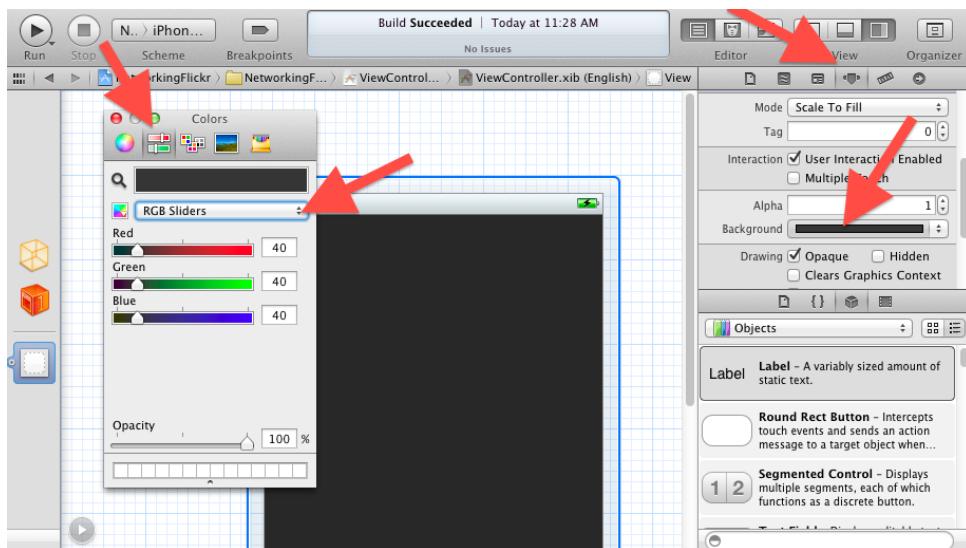



Figura 6.7: Mudando a cor de fundo do componente

Em seguida, adicione o componente "Scroll View", disponível na Object Library (`View -> Utilities -> Show Object Library`), arrastando-o para que fique centralizado e ocupe todo o espaço disponível (O Xcode já deverá deixar o componente automaticamente do tamanho da tela, cabendo a você apenas centralizar). Por último, precisamos conectar o *Outlet* do scroll com o código do controller, e também definir o mesmo controller como classe responsável para tratar dos eventos que o componente irá gerar. Em outras palavras, vamos tornar a classe ViewController como sendo o `delegate` do Scroll View.

Delegates

Delegate é um padrão de projeto no qual um objeto *delega* o trabalho a ser feito para algum outro objeto (o delegate propriamente dito). Uma das principais vantagens é que a classe pode delegar trabalho para algum outro lugar, melhorando a separação de responsabilidades e sem ter que ficar amarrada a uma determinada implementação. Delegates são utilizados frequentemente no desenvolvimento com Objective-C.

Para conectar o outlet o processo é o mesmo de sempre: com o Assistant Editor aberto, selecione o componente no Interface Builder, segure CTRL e arraste para o arquivo ViewController.h ao lado, e crie a connection Outlet com o nome "scroll". Já conectar o evento de delegate envolve alguns passos a mais, mas embora possa parecer complicado numa primeira tentativa, é algo que rapidamente você dominará:

- Primeiramente abra o "Document Outline", clicando na pequena seta que tem no canto inferior esquerdo, no Interface Builder
- Selecione o scroll, então segure CTRL e arraste para cima do item "View Controller", no Document Outline
- No pequeno popup que aparecer, selecione a opção "delegate"
- Feche o Document Outline, clicando na mesma seta usada para abri-lo

Veja a figura 6.8 para referência.

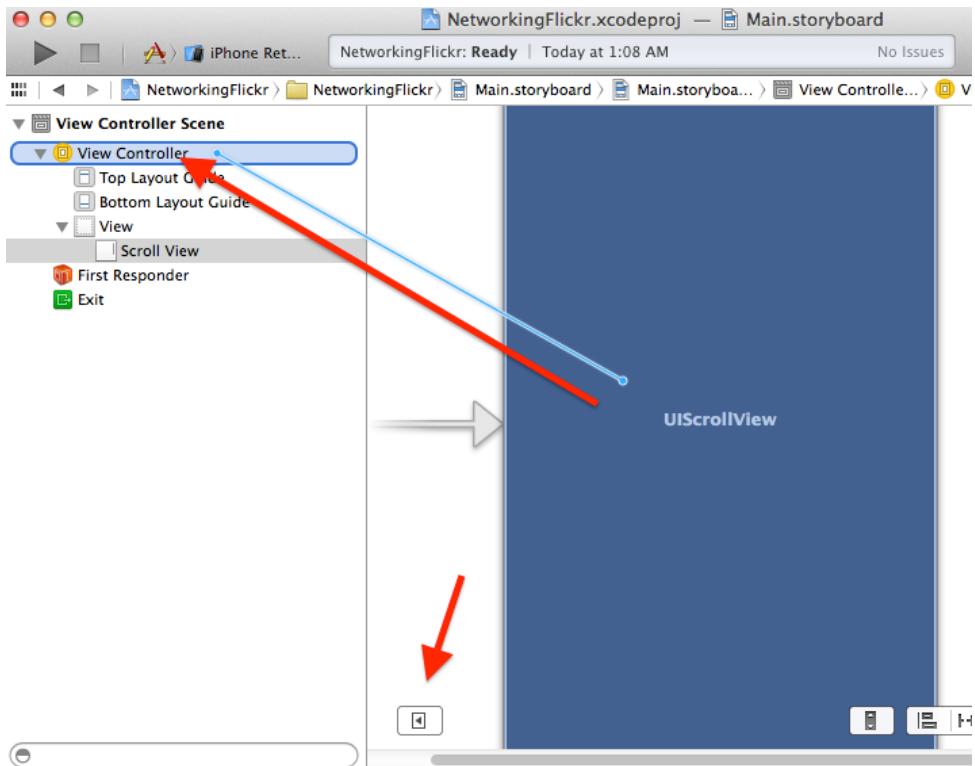



Figura 6.8: Conectando o delegate do scroll view

O que acabamos de fazer foi configurar o scroll view para que use a classe ViewController como sendo a responsável por lidar com os eventos do delegate dele. Para finalizar esta parte da configuração, é necessário modificar o arquivo ViewController.h para que ele possa lidar com os eventos corretamente. Para isso, modifique a declaração `@interface ViewController : UIViewController` para `@interface ViewController : UIViewController<UIScrollViewDelegate>`.

Aproveitando que estamos mexendo no arquivo ViewController.h, declare uma variável de instância do tipo `NSArray` chamada *elementos*, e outra do tipo `NSMutableArray` chamada *imagens*, pois faremos uso delas logo mais. O código do arquivo deverá ficar assim:

```
@interface ViewController : UIViewController<UIScrollViewDelegate>
```

```
@property (retain, nonatomic) NSMutableArray *imagens;
@property (retain, nonatomic) NSArray *elementos;
@property (weak, nonatomic) IBOutlet UIScrollView *scroll;

@end
```

Agora temos diversas tarefas para fazer, de tal maneira que, ao iniciar, o aplicativo conecte-se a API do 500px, baixe as informações das últimas fotos postadas pelos usuários, e mostre-as em um scroll na medida que interagimos com ele. A lista de tarefas pode ser resumida desta maneira:

- Configurar os *blocks* de sucesso e erro do AFNetworking
- Configurar o scroll, para que tenha tamanho suficiente para comportar todas as imagens
- Pré-gerar os componentes de imagens (porém sem carregá-las ainda)
- Código para carregar uma determinada imagem
- Carregar as outras imagens à medida que interagimos com o scroll

6.6 Configurar os *blocks* de sucesso e erro do AFNetworking

A primeira parte do código ficará no método `viewDidLoad`, que é um método declarado na classe `UIViewController` da qual a nossa classe *ViewController* herda (note que os nomes são muito parecidos, cuide para não fazer confusão). Ele é executado automaticamente pelo iOS assim que a view do controller é carregada em memória, portanto é um bom lugar para preparar o controller para uso, como inicializar variáveis, criar outras views auxiliares etc.

Lembre-se dos imports

Não esqueça de importar o arquivo `AFNetworking.h`, através da instrução `#import "AFNetworking.h"` logo no início do arquivo `ViewController.m`

```
1  -(void) viewDidLoad
2  {
3      [super viewDidLoad];
4
5      NSString *url = @"http://bit.ly/livroios-500px";
6      AFHTTPRequestOperationManager *manager =
7              [AFHTTPRequestOperationManager manager];
8
9      manager.responseSerializer = [AFJSONResponseSerializer serializer];
10
11     [manager GET:url parameters:nil
12         success:^(AFHTTPRequestOperation *operation, id json) {
13             _elementos = json[@"photos"];
14             [self mostraMensagem:[NSString
15                 stringWithFormat:@"%d imagens encontradas",
16                     _elementos.count]];
17
18             if (_elementos.count > 0) {
19                 [self inicializaScroll];
20             }
21         }
22         failure:^(AFHTTPRequestOperation *operation, NSError *error) {
23             [self mostraMensagem:[NSString stringWithFormat:@"Erro: %@",
24                 [error localizedDescription]]];
25         }
26     ];
27 }
```

A linha 9 configura o request para lidar com dados JSON, utilizando uma classe própria para isso do AFNetworking, a `AFJSONResponseSerializer`. Na linha 12 declaramos o bloco de sucesso, sendo que o último parâmetro — `id json` — é o que conterá os dados do JSON em si, porém já convertidos para um *NSDictionary* pelo AFNetworking. Esta é a parte em que usar `AFJSONResponseSerializer` é vantajoso, pois caso contrário teríamos que lidar manualmente com alguma biblioteca JSON. Na linha 13 pegamos todos os itens retornados pelo 500px, e guardamos para uso posterior na classe.

Como saber o que terá no NSDictionary?

Dois detalhes importantes: primeiro, a declaração do bloco *success* passa o parâmetro "json" como sendo do tipo `id`, que é um tipo coringa. Se você conhece Java ou C#, é como se fosse o "Object" destas linguagens. Além disso, a documentação do AFNetworking não diz nada explicitamente que o tipo passado será um NSDictionary, porém nos testes realizados foi este o caso, então assume-se que ao menos será um tipo compatível. Em segundo lugar, uma maneira fácil de descobrir a estrutura dos dados no dicionário (que foi a usada a primeira vez que utilizei a biblioteca) é simplesmente jogar o conteúdo no console, através de `NSLog(@"%@", json)`. Como você pode ver, não há nenhuma "mágica" envolvida, apenas um pouco de "programação orientada à tentativa e erro"

O block *failure*, na linha 22, simplesmente mostra a mensagem de erro ao usuário, caso a requisição falhe por qualquer motivo.

6.7 Configurar o scroll e pré-gerar os componentes de imagens

O componente `UIScrollView` que adicionamos no arquivo Main.storyboard foi criado apenas com as configurações padrão, e agora vamos definir o tamanho da área em que o usuário poderá navegar, assim como adicionar as outras views que serão responsáveis por mostrar as imagens. Uma das propriedades mais importantes do UIScrollView é a `contentSize`, que especifica a largura e altura do conteúdo e que, dependendo dos valores informados e do tamanho da tela, permite que o usuário faça scroll para cima ou para baixo. O tipo da propriedade `contentSize` é *CGSize*, que é uma estrutura que comporta largura e altura, com ponto flutuante (`float`), e é relativa ao tamanho do *frame* do scroll. Portanto, se o scroll tem 320x480 de tamanho, um *contentSize* com as mesmas dimensões não permitirá a rolagem do conteúdo, porém se o *contentSize* for de, digamos, 3200x500, será possível rolar uma grande quantidade na horizontal, e um pouco na vertical. Simples assim.

```
1 -(void) inicializaScroll {
2     float largura = self.scroll.bounds.size.width;
```

```
3     float altura = self.scroll.bounds.size.height;
4
5     self.scroll.contentSize = CGSizeMake(largura * _elementos.count,
6                                         altura);
7     self.scroll.pagingEnabled = YES;
8
9     _imagens = [[NSMutableArray alloc] init];
10
11    // Cria todos os lugares onde uma imagem pode
12    // aparecer, para facilitar as coisas na hora
13    // de carregar a imagem de fato do Flickr
14    int indice = 0;
15
16    for (NSDictionary *item in _elementos) {
17        CGRect posicao = CGRectMake(indice++ * largura, 0, largura,
18                                    altura);
19        UIImageView *img = [[UIImageView alloc] initWithFrame:posicao];
20
21        [_scroll addSubview:img];
22        [_imagens addObject:img];
23    }
24
25    // Adiciona a primeira imagem só para não ficar com a tela vazia
26    [self carregaImagemRemota:0];
27 }
```

O código da linha 5 faz exatamente o que foi descrito no parágrafo anterior, definindo a área de rolagem horizontal como sendo o tamanho do componente de scroll pelo número de imagens que temos. Além disso, queremos que o usuário possa rolar o conteúdo precisamente imagem por imagem, dando aquele efeito de que um simples arrastar já passe para a próxima imagem. Isso é feito com o código da linha 6. Nos seus testes, mude o valor da propriedade `pagingEnabled` para *NO* e veja a diferença no comportamento da rolagem.

O `NSMutableArray` iniciado na linha 8 será utilizado para termos acesso rápido e fácil a todos os componentes de imagens criados mais abaixo no método. Existem outras maneiras de obter o mesmo resultado sem precisar de uma variável auxiliar como estamos fazendo este caso, porém a abordagem mostrada neste exemplo é bastante prática e não tem impactos significativos no uso da memória, pois apenas guardaremos referências.

Em iOS, sempre que você quiser mostrar uma imagem ao usuário, fará uso do componente `UIImageView`. Este componente, assim como todas *views*, precisa ser posicionado de maneira absoluta — não existe o conceito de posicionamento relativo em iOS. O código da linha 16 cria a posição onde cada imagem ficará, e na linha 17 é criada a imageview em si. Se não tivéssemos a conta " `indice++ * largura`", todas as imagens ficariam no mesmo lugar, se sobrepondo. A figura 6.9 demonstra este procedimento.

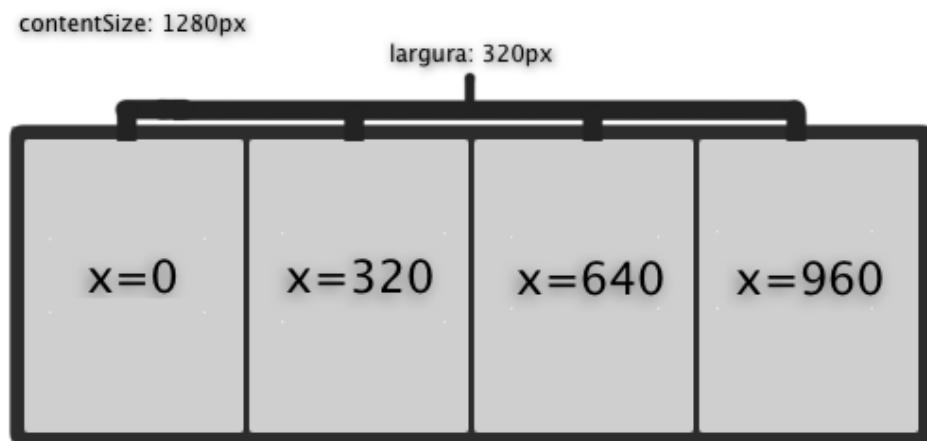



Figura 6.9: Representação gráfica do posicionamento das imagens no componente scroll

Como queremos que as imagens sejam roladas através do nosso componente *scroll* as adicionamos como filhas do mesmo, conforme o código da linha 19.

6.8 Carregar uma determinada imagem

No passo anterior configuramos o UIScrollView e já adicionamos todos os componentes que irão carregar as imagens do 500px, deixando o terreno preparado para o momento de mostrar cada uma. Embora poderíamos criar cada `UIImagemView` somente quando o usuário chegasse no ponto do scroll em que ela fosse necessária, isso demandaria um pouco mais de trabalho, além de correr o risco de causar aquele efeito de "travada" na interface. Porém, como não vamos mostrar milhares de imagens, pré-gerar facilita bastante as coisas.

```
1  -(void) carregaImagemRemota:(int) indice {
2      NSDictionary *item = _elementos[indice];
3      NSDictionary *imageInfo = item[@"images"][0];
4      NSString *url = imageInfo[@"url"];
5  
6      NSLog(@"Carregando a URL %@", url);
7  
8      UIImageView *img = _imagens[indice];
9      img.contentMode = UIViewContentModeScaleAspectFit;
10     [img setImageWithURL:[NSURL URLWithString:url]];
11 }
```

O código da listagem acima recebe como argumento o índice da imagem a ser carregada, relativo ao conteúdo do array *imagens*, pega a URL do dicionário criado pelo AFNetworking lá no primeiro passo, e solicita o download e posterior exibição da imagem na tela. Na linha 8 pegamos a referência ao UIImageView criado no passo anterior, e informamos através da propriedade `contentMode` que a imagem deve ser renderizada de tal maneira que apareça por completo, porém respeitando as dimensões da tela e as devidas proporções. Sem esta propriedade, a imagem teria um aspecto "esticado", fora das proporções originais.

Uma parte interessante deste código está na linha 10, que utiliza mais um recurso do AFNetworking através do método `setImageWithURL:`. Este método não existe na API padrão do componente UIImageView, mas o AFNetworking — por meio do recurso de "*Categorias*", o qual é explicado em detalhes no capítulo sobre Objective-C — adiciona esta funcionalidade. Ela se encarrega de fazer o download da imagem em segundo plano (ou em "background", no termo mais comum) e, caso ela já tenha sido baixada, de utilizar o cache local da mesma. Dessa forma, últimas chamadas a `setImageWithURL:` com um mesmo endereço farão apenas um único download. Isso é uma grande mão na roda, pois se fossemos ter que fazer isso manualmente, o trabalho seria grande.

Importar a biblioteca auxiliar de imagens

Para que o código compile corretamente é necessário importar algumas classes adicionais que vem junto com o AFNetworking. Vá ao menu "*File -> Add files to VisualizadorImagens*" e localize a pasta "UIKit+AFNetworking" no diretório onde descompactou a biblioteca anteriormente.

Clique em "Add" e em seguida adicione a seguinte linha logo no início do arquivo `ViewController.m`:

```
#import "UIImageView+AFNetworking.h"
```

6.9 Carregar as outras imagens à medida que interagimos com o scroll

A última parte do código é responsável por carregar as demais imagens à medida que navegamos pelo scroll. O método `scrollViewDidScroll:` é um método especial, especificado no *UIScrollViewDelegate* que declaramos no arquivo ViewController.h anteriormente. O UIScrollViewDelegate tem diversos métodos que podem ser implementados, como início e fim de zoom, animação e aceleração. O comportamento do *scrollViewDidScroll* é ser executado diversas vezes, permitindo por exemplo que alguma ação (como carregar a imagem) seja feita *enquanto* o scroll está sendo rolado. Contudo, optamos por fazer esta tarefa apenas quando o usuário estiver "parado" em uma determinada posição:

```
-(void) scrollViewDidScroll:(UIScrollView *)scrollView {
    int x = (int)self.scroll.contentOffset.x;
    int largura = self.scroll.frame.size.width;

    // Somente carrega a próxima imagem
    // caso o scroll tenha parado em uma página
    if (x % largura == 0) {
        int pagina = x / largura;
        [self carregaImagemRemota:pagina];
    }
}
```

A propriedade `contentOffset`, utilizada na linha 1, contêm a posição *x* e *y* em que a rolagem do scroll se encontra. Ela é do tipo `CGPoint`. Caso queira visualizar em detalhes os valores desta propriedade, você pode jogar no console desta forma:

```
NSLog("contentOffset: %@",
        NSStringFromCGPoint(self.scroll.contentOffset));
```

Rode o aplicativo (`Command + R`) e navegue pelas fotos. O código completo do aplicativo encontra-se na pasta *VisualizadorImagens*.

6.10 Faça seu aplicativo funcionar em todas as orientações

Um dos problemas que o nosso aplicativo tem é que, se você tentar girá-lo para a orientação landscape (ou "horizontal", como muitas pessoas utilizam, especial leigos), verá que as coisas simplesmente ficam totalmente tortas, conforme mostra a figura 6.10.

Figura 6.10: Aplicativo com problemas para funcionar em landscape

Simulando funcionalidades de um dispositivo real

O iOS Simulator que vem com Xcode funciona surpreendentemente bem, sendo inclusive possível simular diferentes orientações, Geolocalização e alertas de memória. Você encontra diversas opções no menu *Hardware* do simulador. Eis algumas comumente utilizadas:

- **Girar a tela para a esquerda:** `Command + Seta para esquerda`
- **Girar a tela para a direita:** `Command + Seta para direita`
- **Chacoalhar:** `CTRL + Command + Z`
- **Tela em escala 100%:** `Command + 1`
- **Tela em escala 75%:** `Command + 2`
- **Tela em escala 50%:** `Command + 3`

Para solucionar o problema precisamos reorganizar as imagens depois que o aplicativo mudar de orientação, posicionando-as no lugar correto. Isso é necessário porque, quando iniciamos o aplicativo pela primeira vez, ele abre em portrait (ou "vertical"), e tanto o componente `UIScrollView` quando cada `UIImageView` nele adicionadas levam em consideração o tamanho da tela. Ao mudar a orientação, o tamanho da tela também muda, e temos que lidar com isso caso a caso.

Vamos precisar de três coisas: uma variável de controle para sabermos qual é o índice da imagem sendo visualizada no momento, permitir a rotação para todas orientações, e lidar com o evento de rotação do dispositivo. Para a primeira parte, declare uma variável do tipo `int` no arquivo `ViewController.h` chamada `paginaAtual`. Em seguida, salve nela o valor passado ao método `carregaImagemRemota:`, conforme o pedaço de código abaixo:

```
1 -(void) carregaImagemRemota:(int) indice {
2     // Armazena o índice da imagem atual para usar no evento de rotação
3     paginaAtual = indice;
4
```

```
5     // Restante do código já existente no método
6     // ...
7 }
```

Para permitir rotação para todas as orientações, o método "shouldAutorotate:" precisa retornar YES, conforme abaixo:

```
- (BOOL)shouldAutorotate {
    return YES;
}
```

Agora precisamos reorganizar as imagens depois que a tela girar, e para isso basta sobrescrever o método didRotateFromInterfaceOrientation::

```
1  -(void) didRotateFromInterfaceOrientation:
2  (UIInterfaceOrientation)orientacao {
3      float largura = self.scroll.frame.size.width;
4      float altura = self.scroll.frame.size.height;
5      int indice = 0;
6
7      self.scroll.contentSize = CGSizeMake(largura * _elementos.count,
8                                          altura);
9
10     for (UIImageView *img in self.scroll.subviews) {
11         if (img.frame.size.width > 7 && img.frame.size.height > 7) {
12             img.frame = CGRectMake(indice++ * largura, 0, largura,
13                                    altura);
14         }
15     }
16
17     CGPoint novaPosicao = CGPointMake(largura * paginaAtual, 0);
18     [_scroll setContentOffset:novaPosicao animated:NO];
19 }
```

Na linha 7 definimos o novo tamanho do scroll, já que a largura e altura são diferentes em cada orientação. Já entre as linhas 9 a 13 iteramos por cada uma das subviews do scroll para colocá-las na nova posição. A condicional da linha 10 não é exatamente uma boa prática, porém para efeitos de simplicidade da exemplificação ela resolve o problema. Mais especificamente, as barras de rolagem que aparecem por padrão na UIScrollView são imagens também, com altura ou largura de 7 pixels, mas queremos modificar apenas as fotos que vieram do Flickr. Na linha 10 calculamos a nova posição das fotos.

Por fim, o código das linhas 15 e 16 colocam o `scroll` no lugar correto, levando em consideração o índice da última imagem visualizada.

Rode o aplicativo novamente e gire a tela (`Command + Seta para direita`) para ver o resultado.

Capítulo 7

Trabalhando com tabelas - UITableView

Apresentação de dados tabulares é uma das tarefas mais comuns que existem em qualquer plataforma, e no caso de iOS, sistemas inteiros podem ser desenvolvidos em cima desta forma de apresentação de dados, através da classe `UITableView`. Podemos utilizar uma table view para as mais variadas tarefas:

- Lista de países
- Favoritos
- Receitas
- Opções de configuração
- Lista de contatos
- Números de telefone

- Tarefas a serem realizadas

A API deixa relativamente aberto o que podemos fazer com ela, embora a princípio não haja necessidade de muita customização — até porque precisamos levar em conta que o usuário estará usando um dispositivo móvel com os dedos, e não com um mouse. Em suma, a table view disponibiliza suporte a conteúdo adicionado em diversas linhas, que podem opcionalmente serem separadas em seções. Os próprios iDevices contam com diversos aplicativos construídos em cima de table views: *Settings*, *Contacts*, *Reminders*, favoritos do Safari, lista de álbuns do *Photos* e o próprio aplicativo da *App Store*. A figura 7.1 mostra alguns exemplos da `UITableView`.

Figura 7.1: Diversas formas da UITableView

Apesar de ter *table* no nome, o componente em si não atua como uma tabela de diversas linhas e colunas, mas sim apenas de diversas linhas, salvo pequenas customizações que podem ser feitas. Certamente é possível criar células totalmente especializadas, embora isso não seja muito comum por questões de usabilidade. Outro ponto importante é que, embora o conceito da `UITableView` seja bastante simples, aprender a usar a API de maneira eficiente leva um certo tempo, como tudo relacionado à programação para iOS. Lembro que as minhas primeiras experiências com este componente não foram das mais agradáveis justamente porque fiquei insistindo em querer entendê-lo e usá-lo à força, o que foi algo um tanto traumático. Conheça-o aos poucos, como faremos aqui.

O código-fonte deste capítulo está disponível nas pastas “`TableViewSimpleContactList`”, “`TableViewDelete`” e “`TableViewMultipleDelete`” (lembrando que o endereço do site com os

códigos está na introdução do livro).

7.1 Criando a primeira table view - conceitos e exemplo

Ok, se você não desistiu de ler, então ainda há esperança. Lembre-se sempre: a curva inicial de aprendizado da table view é um pouco íngreme, mas depois tudo fica bastante claro e fluído. Justamente por este motivo é muito importante que não pule esta pequena parte conceitual (afinal, o ávido aprendiz quer ver algo funcionando logo, certo?).

Para funcionar, a table view precisa sempre que dois tipos de delegates sejam implementados: O `UITableViewDataSource`, o qual provê os dados e informações estruturais gerais para a `UITableView`, e `UITableViewDelegate`, o qual é responsável por lidar com eventos de integração e customizações de cada linha. As principais responsabilidades de cada um são:

A fonte de dados - UITableViewDataSource

O data source é o protocolo que fornece os dados de fato para a table view, e o único que tem alguns métodos de implementação obrigatória pelo compilador.

- Número de seções da tabela
- Número de linhas (obrigatório)
- Eventos de reordenação, inserção e seleção de dados
- Título do cabeçalho e rodapé
- Fornecer o conteúdo em si de uma determinada linha (obrigatório)

Configurações e ações - UITableViewDelegate

Embora o `UITableViewDelegate` não tenha nenhum método de implementação obrigatória, é com ele que de fato interagimos com a table view em si.

- Altura da linha
- Lidar com interação (toque) de uma determinada linha

- Lidar com a edição de alguma linha

Estas são os principais eventos que precisamos implementar, independentemente do tipo de conteúdo que será apresentado na table view, pois eles são os pilares para o correto funcionamento. Não espero que você consiga se lembrar sempre de cabeça quais são as assinaturas de todos os métodos que devem ser implementados para colocar uma table view para funcionar — eu mesmo não lembro de tudo até hoje. O importante é você saber o que deve ser feito, como fazer e, talvez o mais importante de tudo, saber *onde* encontrar a documentação no caso de dúvidas.

A principal intenção do primeiro exemplo no qual trabalharemos — uma lista de contatos — é focar na forma de criação e organização do código necessário para ter a `UITableView` funcionando. Se em algum momento você pensar "... mas isso é muito código para pouca coisa...", lembre-se que a table view precisa de um conjunto mínimo de informações para funcionar, independentemente se mostraremos um único registro ou um milhão. A figura 7.2 mostra a aplicação depois de finalizada.

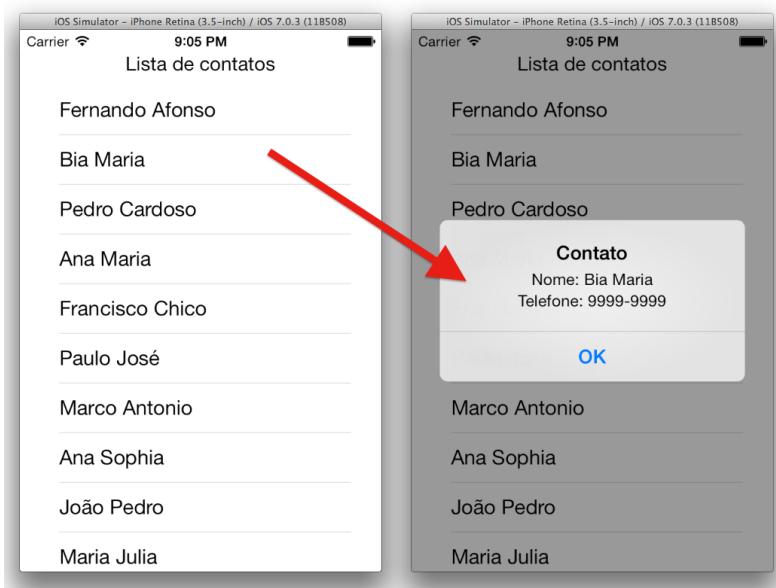



Figura 7.2: Primeiro exemplo com UITableView

7.2 O aplicativo de lista de contatos

Começaremos com um novo exemplo, para depois você praticar com o nosso catálogo de empresas.

O aplicativo que criaremos a seguir consiste em uma hipotética lista de contatos, que mostra o nome e o telefone de um determinado item quando algum elemento da tabela recebe o evento de toque. Para começar, crie um novo projeto do tipo *Simple View Application* chamado "TableViewSimpleContactList", utilizando as opções padrão. Feito isso, siga os passos abaixo:

1) Abra o arquivo `Main.storyboard`;

2) Insira um Label centralizado no topo, com o texto "Lista de contatos";

3) Adicione o componente "Table View" logo abaixo do label, redimensionando-o para ocupar o resto do espaço da tela;

4) Com o *Assistant Editor* aberto (View -> Assistant Editor -> Show Assistant Editor), clique em cima do componente table view, segure CTRL e arraste o mouse para o arquivo `ViewController.h`, inserindo uma *Connection* do tipo `Outlet` chamada `tabelaContatos` (*obs: se ocorrer do Xcode mostrar o arquivo* `.m`*, certifique-se primeiro de alternar para o* `.h` *utilizando a barra superior. Você deverá ver o nome do arquivo nela*);

5) Agora clique com o botão direito em cima do componente table view para abrir o menu de opções, e arraste a bolinha dos outlets `datasource` e `delegate` para o ViewController, que é representado por um círculo amarelo. Veja a figura 7.3 para referência;

6) Para que a nossa classe `ViewController` receba os eventos da table view, é necessário ainda especificar os protocolos `UITableViewDelegate` e `UITableViewDataSource` na declaração da `@interface`;

7) Por último, precisamos de um array para armazenar os contatos que serão carregados de um arquivo de configuração. Para isso, declare um membro de instância do tipo `NSMutableArray` chamado `contatos`.

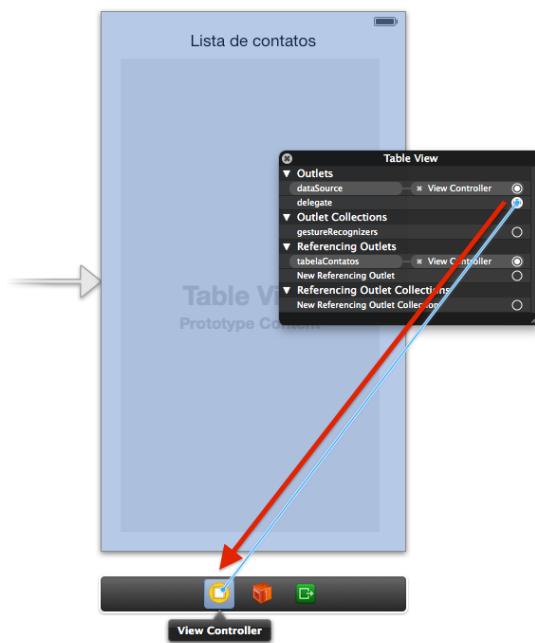



Figura 7.3: Conectando as propriedades datasource e delegate

Veja na listagem abaixo como deverá ficar o seu arquivo `ViewController.h`:

```
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController<UITableViewDataSource,
UITableViewDelegate> {
    NSMutableArray *contatos;
}
@property (weak, nonatomic) IBOutlet UITableView *tabelaContatos;

@end
```

7.3 Carregando os contatos a partir de um arquivo plist

A `UITableView` não impõe quaisquer restrições de onde os dados deverão vir, o que é excelente. Ela apenas pede "*por favor, me forneça os dados da linha X*", e aí

fica a nosso cargo buscar tal informação onde quer que seja. No caso do aplicativo de lista de contatos, as informações estão em um arquivo *plist*, ou "Property List" (particularmente pronuncio como "pê list"), que é um formato relativamente comum em aplicações Mac e iOS para armazenar coisas simples. Na prática, nada mais é que um documento XML com algumas tags específicas. Sua utilidade é como a de um dicionário (chave e valor), com a diferença de que é possível especificar o tipo de dados.

Crie o arquivo `contatos.plist` em *File* -> *New* -> *File...* -> *Resource* -> *Property List*, e abra-o no Xcode. Existem três colunas: *Key*, que obviamente é o nome da chave do dicionário, *Type* para informar o tipo de dados (dicionário, array, número, string, data, booleano ou conteúdo "genérico"), e *Value*, que é o valor da chave de fato.

Logo após criar o arquivo, você verá que existe uma chave chamada "Root" do tipo "Dictionary", e o conteúdo que formos adicionar deverá ser filho desta chave. Clique com o botão direito e selecione a opção " `Add Row`", e em seguida nomeie "*contatos*" e selecione " `Array`" como "*Type*". Cada um dos elementos deste array será outro dicionário contendo as chaves "*nome*" e "*telefone*", os quais iremos ler e organizar em uma estrutura de dados especializada na hora em que o aplicativo for iniciado.

Clique com o botão direito em cima de "contatos" e selecione *Add Row*. A *Key* do item pode ser qualquer coisa (usei "Item 0", "Item 1" etc.), enquanto que o *Type* deve ser configurado para `Dictionary`. Insira duas chaves do tipo `String`, uma para o nome e outra para o telefone do contato. Repita a operação até cansar. A figura 7.4 mostra como deverá ficar o seu arquivo.

Key	Type	Value
contatos	Array	(12 items)
Item 0	Diction...	(2 items)
nome	String	Ana Sophia
telefone	String	1111-1111
Item 1	Diction...	(2 items)
nome	String	Maria Julia
telefone	String	2222-2222
Item 2	Diction...	(2 items)
nome	String	João Pedro
telefone	String	3333-3333
Item 3	Diction...	(2 items)
Item 4	Diction...	(2 items)
Item 5	Diction...	(2 items)
Item 6	Diction...	(2 items)
Item 7	Diction...	(2 items)
Item 8	Diction...	(2 items)
Item 9	Diction...	(2 items)
Item 10	Diction...	(2 items)
Item 11	Diction...	(2 items)

Figura 7.4: Organização do arquivo contatos.plist

Corrigindo o aninhamento

Caso os elementos não estejam sendo criados na hierarquia esperada, é possível aninhá-los com o item de menu "*Shift Row Right*" para tornar o elemento filho do que se encontra logo acima, e "*Shift Row Left*" para a operação contrária. Clique com o botão direito no elemento e selecione a opçõa desejada.

Para carregar o conteúdo do arquivo usamos a classe `NSDictionary`, e na prática isso já seria suficiente para trabalharmos com a table view. Porém, dicionários com muitos dados são chatos de trabalhar e muito suscetíveis a erros de digitação. O ideal, portanto, é utilizar uma estrutura de dados especializada, e "transformar" o dicionário em instâncias desta estrutura. Crie uma nova classe chamada `Contato` com duas propriedades do tipo `NSString` (já que as chaves no property list são "strings") chamadas `nome` e `telefone`, conforme o código abaixo do `Contato.h`:

```
#import <Foundation/Foundation.h>
```

```
@interface Contato : NSObject

-(id) initWithNome:(NSString *) nome andTelefone:(NSString *) telefone;

@property (nonatomic, retain) NSString *nome;
@property (nonatomic, retain) NSString *telefone;

@end
```

E aqui do `Contato.m`:

```
#import "Contato.h"

@implementation Contato

-(id) initWithNome:(NSString *) nome andTelefone:(NSString *) telefone {

    if ((self = [super init])) {
        self.nome = nomeInicial;
        self.telefone = telInicial;
    }

    return self;
}

@end
```

Já temos pronto a lista de contatos em um arquivo próprio e a estrutura de dados `Contato` para representar no código cada registro daquele arquivo. O próximo passo consiste, portanto, em juntar estes dois pedaços, carregando o conteúdo do arquivo `contatos.plist` em memória, armazenando os registros na variável `contatos`. Confira abaixo o método `loadContacts`, que deve ser implementado no arquivo `ViewController.m`:

```
1 -(void) loadContacts {
2     NSString *plistCaminho = [[NSBundle mainBundle]
3         pathForResource:@"contatos"  ofType:@"plist"];
4     NSDictionary *pl = [NSDictionary
5                             dictionaryWithContentsOfFile:plistCaminho];
6     NSArray *dados = [pl objectForKey:@"contatos"];
7
```

```
8       contatos = [[NSMutableArray alloc] init];
9
10      for (NSDictionary *item in dados) {
11          NSString *nome = [item objectForKey:@"nome"];
12          NSString *telefone = [item objectForKey:@"telefone"];
13
14          Contato *c = [[Contato alloc] initWithNome:nome
15                  andTelefone:telefone];
16          [contatos addObject:c];
17      }
18  }
```

Obs: não se esqueça de importar o header de Contato, com a instrução `#import "Contato.h"` logo no início do arquivo

Como para fins práticos para o iOS um arquivo Property List é um dicionário, carregá-lo em memória é bastante simples, conforme a linha 4: basta iterar pelos registros transformando-os em `Contato`s.

A classe NSBundle

A classe `NSBundle` contém métodos para encontrar e carregar coisas que façam parte do pacote do seu aplicativo. Por exemplo, o método `pathForResource:` utilizado em `loadContacts` procura o arquivo especificado a partir da raiz do projeto.

Para que o método seja executado ao iniciar o aplicativo, chame-o no método `viewDidLoad`, conforme abaixo:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    [self loadContacts];
}
```

Nome das variáveis no método initWithNome

Algumas pessoas irão reparar que as variáveis do método `initWithNome` tem o mesmo nome das `@property`. Para evitar conflitos em casos como esse, usa-se a instrução `self.nome` e `self.telefone` para referenciar os membros da classe. Algumas pessoas preferem utilizar underline nestas situações, como `initWithNome:(NSString *) _nome andTelefone:(NSString *) _telefone`, mas isso é algo completamente de gosto pessoal.

7.4 Tornando a tabela funcional

Agora que já temos a estrutura geral do aplicativo construída, incluindo a lista de contatos que deverá ser apresentada ao usuário, é hora de fazer as partes específicas da table view, que são três. Teremos de informar quantos itens temos para mostrar, fornecer os dados de cada um dos contatos para cada linha e lidar com o evento de toque em um determinado contato.

Os dois primeiros itens fazem parte do protocolo `UITableViewDataSource`, enquanto o terceiro pertence a `UITableViewDelegate`. E como eu sei isso? Fácil, lendo a documentação. Depois de algum tempo você provavelmente irá se lembrar de memória o que fica em qual lugar, portanto não se preocupe demais em querer decorar tudo — apenas lembre-se sempre: se não souber onde está algo, acesse a documentação.

Acessando a documentação da API pelo Xcode

Existe uma maneira bem fácil e prática de acompanhar e acessar a documentação da API do iOS diretamente pelo Xcode, através do inspetor *Quick Help*, conforme mostrado na figura 7.5. Para habilitar, vá no menu *View -> Utilities -> Show Quick Help Inspector*, e aparecerá um painel com uma breve descrição da classe onde o cursor do mouse se encontra, no editor de códigos.

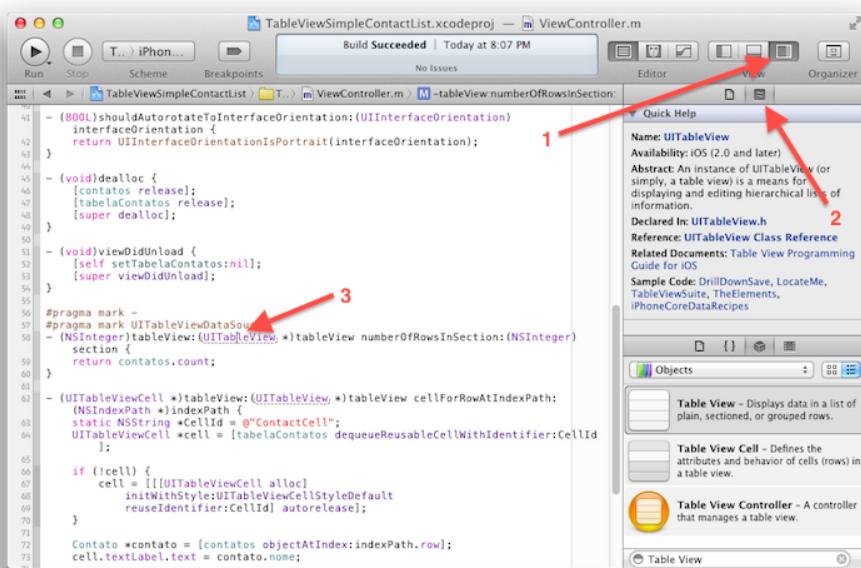



Figura 7.5: Funcionamento do painel de ajuda rápida

7.5 Informando a quantidade de itens que temos

A primeira coisa a fazer é informar quantas linhas a table view deverá ter, conforme o código a seguir:

```
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {

    return contatos.count;
}
```

7.6 Exibindo dados em cada linha

O componente `UITableView` utiliza uma abordagem um pouco diferente para exibir registros, na qual — ao invés de montarmos as linhas com tudo o que a tabela deverá conter — é preciso entregar registro por registro de maneira indireta, deixando a cargo da table view o trabalho de construir as linhas e gerenciar memória e performance. Em outras palavras, é a table view quem solicita o que deve exibir em uma determinada linha, fazendo uso do método `cellForRowAtIndexPath` definido no protocolo `UITableViewDataSource` (lembre-se: "datasource" significa, literalmente, "fonte de dados" em português).

O código abaixo faz exatamente isso: implementamos o método do datasource, o qual nos passa o índice do elemento desejado, de tal forma que a célula da table view contenha como texto o `nome` do contato.

```
1  - (UITableViewCell *)tableView:(UITableView *)tableView
2      cellForRowAtIndexPath:(NSIndexPath *)indexPath {
3
4      static NSString *CelulaContatoCacheID = @"CelulaContatoCacheID";
5      UITableViewCell *cell = [self.tabelaContatos
6          dequeueReusableCellWithIdentifier:CelulaContatoCacheID];
7
8      if (!cell) {
9          cell = [[UITableViewCell alloc]
10             initWithStyle:UITableViewCellStyleDefault
11             reuseIdentifier:CelulaContatoCacheID];
12     }
13
14     Contato *contato = [contatos objectAtIndex:indexPath.row];
15     cell.textLabel.text = contato.nome;
16
17     return cell;
18 }
```

Rode o aplicativo (`Command + R`, ou *Product* -> *Run*) e veja os contatos apare-

cerem na tabela. Toque em alguns registros também. Acontece alguma coisa? Veremos logo mais como implementar esta parte.

Existem alguns conceitos importantes na forma de construção deste código, sendo o principal deles em relação a performance. Para entender melhor, considere que a `UITableView` é um componente que pode ser utilizado para uma infinidade de situações, apresentando ao usuário uma quantidade ilimitada de elementos. Como o aplicativo roda em dispositivos móveis onde os recursos de hardware são mais limitados — especialmente memória —, é de vital importância que façamos uso de mecanismos para *reaproveitar* algumas coisas. Embora fosse possível fazer o código da listagem anterior de maneira mais simples, o mesmo pode eventualmente ser péssimo em termos de performance.

Testando questões de performance no simulador versus no device

O iOS simulator é uma excelente ferramenta para auxiliar o desenvolvimento de aplicativos, pois evita que tenhamos que fazer deploy diretamente em algum dispositivo. Porém, algumas coisas — como o compasso ou eventos de inclinação — não podem ser simuladas nele, e outras, como memória e processador, são irreais, uma vez que o simulador roda sob o Mac OS X. Portanto, a única maneira de saber como o aplicativo se comporta com muita informação e até mesmo para saber se as animações estão fluídas, é testar diretamente no device.

Para auxiliar neste quesito, a `UITableView` nos fornece a opção de reutilizar as `UITableViewCell`s que pertençam a uma mesma categoria de dados (o que deve representar a maior parte dos casos em seus aplicativos), o que é feito nas linhas 5 e 6 com o método `dequeueReusableCellWithIdentifier:`. Ele recebe um identificador que é utilizado internamente para pegar *cells* de um cache sempre que possível. No caso do nosso aplicativo, o identificador é aquele representado pelo código da linha 4, que chamamos de *CelulaContatoCacheID*, que é um valor arbitrário, sem nenhum significado especial.

Caso você esteja se perguntando se existe a possibilidade de ocorrer algum tipo de problema com esta abordagem de reutilização de células, a resposta é "talvez". Ou seja, se partirmos do princípio que ao invés de criar 1000 instâncias da célula a tabela cria apenas 20 e as reutiliza para todo o conjunto de dados, é responsabilidade

do programador garantir que todas as propriedades relevantes sejam preenchidas, caso contrário o valor definido por quem utilizou a célula por último será exibido. Faça um teste e comprove você mesmo.

Na linha 10 iniciamos a célula com o estilo padrão (`UITableViewCellStyleDefault`), o qual mostra uma linha de texto apenas, que é aquele atribuído na linha 15. As outras opções são as listadas abaixo, e na figura 7.6 há uma demonstração de cada um dos estilos.

- `UITableViewCellStyleSubtitle`: Título em negrito alinhado no canto superior esquerdo, e um subtítulo cinza alinhado no canto inferior esquerdo, acessível pela propriedade `detailTextLabel`
- `UITableViewCellStyleValue1`: Texto na cor preta alinhado à esquerda, e um outro texto na cor azul alinhado à direita, acessível pela propriedade `detailTextLabel`
- `UITableViewCellStyleValue2`: Texto da propriedade `textLabel` à esquerda e na cor azul, e o valor da propriedade `detailTextLabel` à direita na cor preta

Figura 7.6: Os diferentes estilos pré-configurados de uma UITableViewCell

7.7 Permitindo interação com os itens da tabela

Toda vez que o usuário toca em uma linha da table view, o método `didSelectRowAtIndexPath:` é disparado, dando-nos a chance de realizar alguma ação — que no nosso caso será mostrar uma mensagem utilizando `UIAlertView`. Como muitos dos outros eventos da table view, este método recebe o índice da linha selecionada, que usaremos para pegar o respectivo registro na variável `contatos`:

```
1  - (void)tableView:(UITableView *)tableView
2  didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
3  
4      Contato *contato = [contatos objectAtIndex:indexPath.row];
5      NSString *msg =
6          [NSString stringWithFormat:@"Nome: %@\nTelefone: %@",
7          contato.nome, contato.telefone];
8  
9      UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Contato"
10         message:msg
11         delegate:nil
12         cancelButtonTitle:@"OK"
13         otherButtonTitles:nil];
14     [alert show];
15 
16     [self.tabelaContatos deselectRowAtIndexPath:indexPath animated:YES];
17 }
```

O código da linha 13 faz com que a linha automaticamente perca a seleção após a interação (remova a linha e veja o que acontece). Rode novamente o aplicativo e interaja com os registros.

7.8 Removendo elementos da table view

Em aplicativos iOS, uma operação que os usuários estão acostumados a fazer é realizar o movimento de *swipe* da direita para esquerda (ou, tecnicamente um *swipe left*), que é aquele movimento de deslizar rapidamente o dedo no eixo horizontal. Tanto por conveniência quando para manter um comportamento padrão nos aplicativos a `UITableView` (através do delegate `UITableViewDelegate`) já disponibiliza tal funcionalidade, bastando implemen-

tarmos o método `tableView:commitEditingStyle:forRowAtIndexPath:` conforme abaixo:

```
1 -(void) tableView:(UITableView *)tableView
2 commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
3 forRowAtIndexPath:(NSIndexPath *)indexPath {
4
5     [contatos removeObjectAtIndex:indexPath.row];
6     [self.tabelaContatos reloadData];
7 }
```

O funcionamento é o seguinte: o iOS verifica se existe a implementação deste método na nossa classe, e caso positivo, habilita a operação de *swipe left* para remover alguma linha, invocando o método em si quando o usuário interage com o botão. Veja na figura 7.7 o resultado.

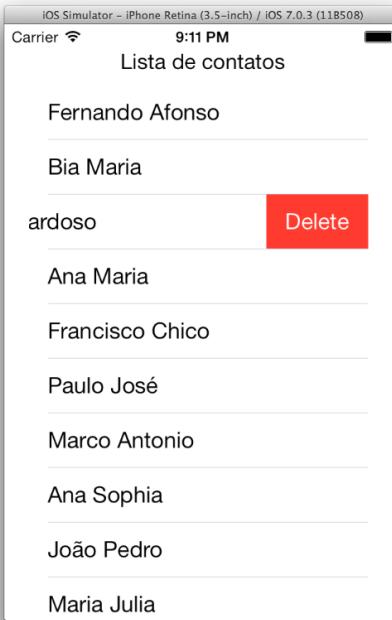



Figura 7.7: Interface padrão do botão deletar

Note que o texto padrão no botão está em inglês. Para especificar outro texto, temos que implementar o método `tableView:titleForDeleteConfirmationButtonForRowAtIndexPath`:

```
-(NSString *) tableView:(UITableView *)tableView
titleForDeleteConfirmationButtonForRowAtIndexPath:
                                (NSIndexPath *)indexPath {
    return @"Remover";
}
```

7.9 Removendo diversas linhas

Uma outra maneira de remover registros é aquele em que o usuário toca em um botão *Editar* e a table view mostra uma série de pequenos botões redondos com um símbolo de subtração, conforme mostrado na figura 7.8. Para obter este resultado, adicione um botão na interface chamado `botaoEditar`, e conecte uma ação a ele chamada `botaoEditarTap`, onde iremos enviar uma mensagem à tabela para que ela entre em modo de edição. Por último, especificamos o tipo de operação a ser feito em cada uma das linhas.

Passos para adicionar o botão

Caso tenha ficado na dúvida, os passos são esses: abra o arquivo `Main.storyboard`, adicione um componente "Button" ao lado do label "Lista de contatos", ative o `Assistant editor`, certifique-se que o arquivo `ViewController.h` esteja selecionado no painel da direita, CTRL+clique e arraste do botão para o arquivo .h para criar um Outlet com o nome de "botaoEditar", e em seguida mais um CTRL+clique e arraste para criar uma Action com o nome "botaoEditarTap".

O código do arquivo "ViewController.h" deverá estar assim:

```
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController<UITableViewDataSource,
UITableViewDelegate> {
    NSMutableArray *contatos;
}
@property (weak, nonatomic) IBOutlet UIButton *botaoEditar;
@property (weak, nonatomic) IBOutlet UITableView *tabelaContatos;
- (IBAction)botaoEditarTap:(id)sender;

@end
```

Confira o código abaixo:

```
1  // Executado quando o usuário toca no botão Editar
2  - (IBAction)botaoEditarTap:(id)sender {
3      if ([self.botaoEditar.titleLabel.text isEqualToString:@"Editar"]) {
4          [self.tabelaContatos setEditing:YES animated:YES];
5          [self.botaoEditar setTitle:@"Pronto"
6                            forState:UIControlStateNormal];
7      }
8      else {
9          [self.tabelaContatos setEditing:NO animated:YES];
10         [self.botaoEditar setTitle:@"Editar"
11                           forState:UIControlStateNormal];
12     }
13 }
```

```
14
15 // Tipo de operação a ser feita. Neste caso em específico,
16 // queremos permitir apenas a remoção de elementos
17 // (é possível inserir novos também)
18 -(UITableViewCellEditingStyle) tableView:(UITableView *)tableView
19 editingStyleForRowAtIndexPath:(NSIndexPath *)indexPath {
20     return UITableViewCellEditingStyleDelete;
21 }
```

O código do método `botaoEditarTap:` não tem muito mistério, sendo que a ação de colocar a table view em modo de edição e sair dele é feito nas linhas 4 e 8 respectivamente. De resto, verificamos o texto do botão para alterar a mensagem mostrada ao usuário. Já na linha 17 informamos explicitamente que a única operação que queremos permitir é a de remoção de elementos — a outra possível é inserir novos registros. O resto do trabalho é feito pelo código visto nos exemplos anteriores.

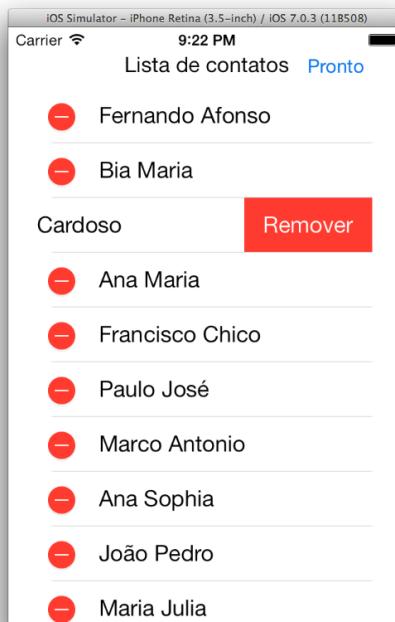



Figura 7.8: Removendo múltiplas linhas

7.10 Crie uma tabela para o nosso catálogo de empresas

Eis um desafio: utilizar os conhecimentos aprendidos até agora para modificar o aplicativo de catálogo de empresas feito no capítulo 3, para que a listagem das empresas seja mostrada em uma table view ao invés de exibir apenas no console. A ideia é que, após clicar no botão salvar, o atual método `mostraCatalogo` apresentado na seção *3.16* instancie um novo controlador, o qual receberá como argumento a lista de empresas disponíveis na variável `catalogo`, e construirá a table view com os dados. Utilize o `UINavigationController` (apresentado no capítulo 4) para ir de um controlador para outro.

Como ponto de partida, você pode seguir os seguintes passos: crie uma nova classe chamada "ExibeCatalogoController" que estenda `UIViewController`, utilizando a opção de criar o arquivo `.xib` conjuntamente. Adicione um componente `UITableView` e conecte-o ao arquivo `ExibeCatalogoController.h`, aproveitando para declarar também uma `@property` para a variável `catalogo` que será passada pela outra classe. Não esqueça de informar os protocolos `UITableViewDataSource` e `UITableViewDelegate`.

O esqueleto mínimo da nova classe deverá se parecer com o código abaixo:

```
// ExibeCatalogoController.h
#import <UIKit/UIKit.h>

@interface ExibeCatalogController :
    UIViewController<UITableViewDelegate, UITableViewDataSource>

@property (weak, nonatomic) IBOutlet UITableView *tabela;
@property (nonatomic, assign) NSArray *catalogo;

@end

// ExibeCatalogoController.m
#import "ExibeCatalogController.h"
#import "Empresa.h"

@implementation ExibeCatalogController
@synthesize tabela, catalogo;

- (NSInteger)tableView:(UITableView *)tableView
```

```
    numberOfRowsInSection:(NSInteger)section {

    return self.catalogo.count;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    return nil;
}
@end
```

Já no método `mostraCatalogo`, chame o novo controlador desta forma:

```
-(void) mostraCatalogo {
    ExibeCatalogController *c = [[ExibeCatalogController alloc] init];
    c.catalogo = catalogo;
    [self.navigationController pushViewController:c animated:YES];
}
```

Boa sorte! Toda informação de que você precisa está nos capítulos apresentados até agora no livro.

Capítulo 8

Trabalhando com reconhecedores de gestos

Diferentemente de aplicativos que são feitos para serem utilizados em computadores com mouse e teclado físicos, a interação com o iOS dá-se unicamente por diversos tipos de toques na tela, indo muito além do que é possível fazer apenas com um mouse. A operação mais padrão e natural é a de tocar uma única vez na tela como se estivéssemos clicando. É algo tão simples que não precisa de maiores explicações — uma criança de 1 ano de idade ou uma pessoa de 95 anos sem afinidade tecnológica se sairá muito bem com esta operação.

Em contrapartida, existem uma série de outras interações que requerem um pouco mais de trabalho para interpretar corretamente, como deslizar o dedo (*swipe*), tocar e segurar (*long press*), movimento de pinça (*pinch*), dois toques (*double tap*) e o que mais a imaginação permitir.

Existem duas maneiras de lidar com eventos de toque: trabalhando com o sistema em mais baixo nível, que permite o máximo de customização (e também requer

muito mais trabalho para implementar), e os reconhecedores de gestos, que são algumas classes prontas para as operações mais comuns.

O código-fonte deste capítulo está disponível nas pastas "GestoSwipe", "TapCirculos", "TremeTreme" e "ViewTouchEventEx1" (lembrando que o endereço do site com os códigos está na introdução do livro).

8.1 Sistema de eventos tradicional

O sistema de eventos tradicional — assim chamava a única abordagem que existia até o iOS 3.2 — consiste no trabalho conjunto de uma série de métodos que, embora mais trabalhosos de implementar, permitem o máximo de customização na maneira como os toques que o usuário faz na tela são interpretados. Todos os métodos pertencem à classe `UIView` (mais precisamente, pertencem a `UIResponder`, de quem `UIView` herda) e podem ser sobrescritos conforme forem necessários. Os principais são:

- `-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event`
- `-(void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event`
- `-(void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event`
- `-(void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event`

O método `touchesBegan` ocorre quando o usuário inicialmente toca a tela, e o `touchesEnded` é disparado quando o usuário remove o dedo. Já `touchesCancelled` é disparado pelo iOS quando alguma coisa ocorre em algum lugar que obriga os eventos de toque a serem cancelados — os motivos variam bastante. Por último, `touchesMoved` serve para lidarmos com os casos onde o usuário arrasta o dedo pela tela.

Interação com componentes comuns do dia a dia

Algo importante a ressaltar é que toda informação deste capítulo aplica-se a views customizadas que construímos. Os componentes padrão do dia a dia, como `UIButton`, `UITableView`, `UITextView` e afins já têm os seus próprios mecanismos para lidar com interações do usuário. As possibilidades de customização que são apresentadas aqui são um mecanismo para casos especiais.

Para melhor demonstrar como lidar com estes eventos, vamos fazer um aplicativo que desenha círculos vermelhos em cada região de toque. Crie um novo projeto chamado "*ViewTouchEventEx1*" e logo em seguida adicione uma nova classe chamada `CirculoView`, que herda de `UIView` (*File* -> *New* -> *File* -> *Cocoa Touch* -> *Objective-C Class* -> *Subclass of UIView*). A razão de criarmos uma view customizada é que os eventos de toque precisam ser sobrescritos e, além disso, para desenhar os círculos é necessário também implementar manualmente o código de desenho.

Como a intenção é desenhar todos os pontos de toque, e não apenas o último, precisamos guardar cada uma das interações. Para isso, usamos um `NSMutableArray` (já que o `NSArray` não pode ser modificado após ser criado), conforme o código abaixo, do arquivo `CirculoView.h`:

```
// CirculoView.h
#import <UIKit/UIKit.h>

@interface CirculoView : UIView {
    NSMutableArray *circulos;
}
@end
```

A diversão ocorre no arquivo `CirculoView.m`. A primeira parte do código consiste em registrar um log de quando o usuário toca na tela (unicamente por questões de visualizar a ordem dos eventos), e registrar o ponto de toque quando o dedo for retirado da tela. Veja o código abaixo:

```
1 // CirculoView.m - Código parcial
2 -(void) touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
3     NSLog(@"Recebido touchesBegan");
4 }
```

```
5
6 -(void) touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
7     if (!circulos) {
8         circulos = [[NSMutableArray alloc] init];
9     }
10
11    NSLog(@"Recebido touchesEnded. Registrando o ponto de toque");
12
13    UITouch *toque = [touches anyObject];
14    CGPoint localizacaoToque = [toque locationInView:self];
15
16    // Precisamos encapsular o CGPoint dentro dentro de um NSValue
17    // o CGPoint não é um objeto propriamente dito
18    [circulos addObject:[NSValue valueWithCGPoint:localizacaoToque]];
19
20    [self setNeedsDisplay];
21 }
```

Por conveniência iniciamos a propriedade `circulos` nas linhas 7 a 9, porém geralmente o mais comum é fazer isso em algum método `init` — mas não existe uma regra, e na prática acaba valendo muito mais a praticidade e tipo de código sendo feito.

Na linha 13 pegamos o objeto de toque em si, representado pela classe `UITouch`. Note que o pedaço correspondente a `[touches anyObject]` é um pouco estranho, mas para este contexto é seguro lê-lo como "retorne o `UITouch` mais conveniente". Uma vez tendo obtido o objeto que representa o toque, precisamos saber em qual parte da tela o usuário colocou o dedo, para desenhar o círculo naquela posição. Isso é feito na linha 14 através do método `locationInView` da `UITouch`, que recebe uma outra view como parâmetro (no caso, a nossa própria view) e retorna uma `struct` do tipo `CGPoint` com a posição convertida para o sistema de coordenadas da nossa view.

A nossa lista `circulos` somente trabalha com objetos, e o `CGPoint`, por ser uma `struct`, acaba sendo incompatível. Para isso, na linha 18 adicionamos a posição obtida na lista `circulos` encapsulando a variável `localizacaoToque` em um `NSValue`, que pode ser considerado como um "objeto coringa" para carregar diversos tipos primitivos em lugares onde apenas objetos são aceitos.

Por último, na linha 20, informamos ao iOS que é necessário redesenhar a view, para que o círculo seja mostrado na tela. Isso é feito no código abaixo:

```
1  -(void) drawRect:(CGRect)rect {
2      CGContextRef contexto = UIGraphicsGetCurrentContext();
3      CGContextSetLineWidth(contexto, 2.0);
4      CGColorRef corFundo = [UIColor redColor].CGColor;
5      CGContextSetFillColor(contexto, CGColorGetComponents(corFundo));
6      int tamanho = 25;
7  
8      for (NSValue *item in circulos) {
9          CGPoint ponto = [item CGPointValue];
10         CGRect regiao = CGRectMake(ponto.x, ponto.y, tamanho, tamanho);
11         CGContextAddEllipseInRect(contexto, regiao);
12         CGContextFillEllipseInRect(contexto, regiao);
13     }
14 
15     CGContextStrokePath(contexto);
16 }
```

O método `drawRect` é o responsável por realizar as tarefas de desenho na nossa view, e o iOS chama-o automaticamente quando necessário. Numa primeira olhada o código completo do método assusta, pois é bastante coisa para pouco resultado, utilizando a API procedural do *Core Graphics* para fazer os desenhos. A parte mais relevante concentra-se entre as linhas 8 e 13, onde iteramos por todos os pontos de toque registrados no método `touchesEnded`, desenhando os círculos na tela.

Como a lista `circulos` contém objetos do tipo `NSValue`, na linha 9 utilizamos o método `CGPointValue` para recuperar o `CGPoint` adicionado quando o usuário interagiu com o aplicativo, e na linha 10 definimos a área, enquanto que o código das linhas 11 e 12 pinta o círculo propriamente dito.

A propriedade userInteractionEnabled

Toda `UIView` contém uma propriedade chamada `userInteractionEnabled`, que quando definida para o valor `YES`, permite a interação do usuário com a view através de eventos de toque. Por padrão este valor é definido para `YES`, porém algumas classes, como a `UIImageView`, definem esta propriedade para `NO`, sendo então necessário habilitá-la manualmente.

Para utilizar a "CirculoView", crie uma nova instância dela no método `loadView` no arquivo "ViewController.m", conforme abaixo:

```
#import "ViewController.h"
#import "CirculoView.h"

@implementation ViewController

-(void) loadView {
    CirculoView *c = [[CirculoView alloc] init];
    c.backgroundColor = [UIColor whiteColor];
    self.view = c;
}

@end
```

O resultado do aplicativo está demonstrado na figura 8.1.

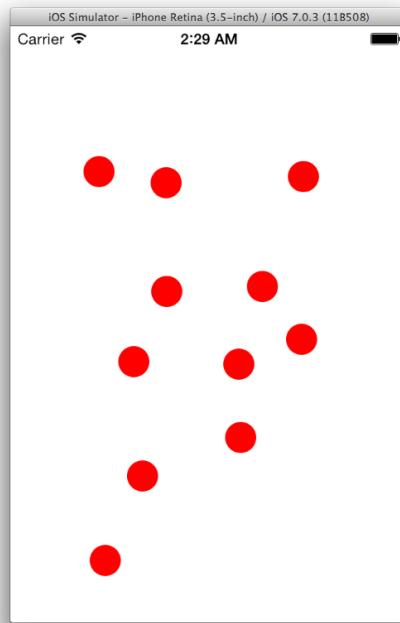



Figura 8.1: Resultado do aplicativo que desenha círculos

8.2 Uma abordagem mais prática: D de gestos

Da mesma forma que os eventos de toque permitem um elevado nível de customização e controle sobre as interações do usuário com a tela, eles podem facilmente se tornar mais complicados de desenvolver e manter do que poderíamos desejar. Para auxiliar neste quesito, a Apple introduzir na API do iOS o conceito de reconhecedores de gestos (do termo em inglês *Gesture Recognizer*), que são representados por um conjunto de classes para as operações mais comuns do dia a dia, além da possibilidade de que os nossos próprios gestos sejam criados (sugestão: funcionalidade especial no seu aplicativo para quem desenhar um disco voador utilizando gestos).

Os seis gestos padrão que vem com o iOS são os seguintes:

- Toque (*tap*): classe `UITapGestureRecognizer`
- Pinça, para zoom (*pinch in* e *pinch out*): classe `UIPinchGestureRecognizer`
- Arrastar (*panning*): `UIPanGestureRecognizer`
- Deslizar (*swipe*): `UISwipeGestureRecognizer`
- Rotacionar: `UIRotationGestureRecognizer`
- Toque longo (*long press*): `UILongPressGestureRecognizer`

Como mencionamos no início do livro, a questão sobre traduzir ou não determinados termos em livros técnicos é sempre polêmica, e no caso dos reconhecedores de gestos (*gesture recognizers*) ainda existe o complicador que muitos dos termos originais se referem às próprias classes e conceitos. Neste capítulo mostraremos sempre que possível o termo em português seguido do termo em inglês, por duas razões: a primeira delas é que os clientes e usuários leigos têm mais facilidade para entender instruções como "faça o movimento de pinça" do que "faça um *pinch*", e isso ajuda muito em reuniões e outras conversas. O segundo motivo — mostrar também o termo em inglês — é para facilitar a identificação dos métodos e classes da API, assim como facilitar na busca por conteúdo na Internet.

Existe uma diferença quase sutil entre os gestos de arrastar (*panning*) e deslizar o dedo (*swipe*) que pode não ficar muito clara no início: a primeira consiste em colocar o dedo na dela e arrastá-lo continuamente, para mover uma view de um lugar para outro, por exemplo. Já o deslizar é um movimento que se assemelha a arrastar, porém

ocorre mais rapidamente e de uma única vez, sendo muito utilizado para "folhear" jornais e revistas e passar de uma imagem para outra na galeria de fotos.

Gestos são sempre associados a alguma view. Quando identificados, o sistema operacional envia mensagens à classe de eventos que foi especificada na criação do gesto. Além disso, são categorizados em dois tipos: contínuos, em que mensagens do evento são enviadas continuamente enquanto o usuário está com o dedo na tela — tais como o gesto de deslizar (*panning*) —, e descontínuos, que ocorrem apenas uma única vez — tais como toque (*tap*) ou deslizar (*swipe*). O evento de toque pode ser utilizado tanto para toque simples (*single tap*) como para duplo (*double tap*) ou mesmo mais: basta especificar a quantidade de toques desejadas na hora de criar a classe.

8.3 Convertendo o exemplo CirculoView para gestos

Crie um novo projeto chamado `TapCirculos` da mesma forma do projeto feito no início deste capítulo, porém adicione a classe `TapCirculoView` desta vez (ao invés da `CirculoView` que foi utilizada anteriormente). O arquivo *TapCirculoView.h* é igual:

```
// TapCirculoView.h
#import <UIKit/UIKit.h>

@interface TapCirculoView : UIView {
    NSMutableArray *circulos;
}
@end
```

A principal mudança será na classe `TapCirculoView`, onde precisamos registrar o evento de toque desejado, e lidar com a mensagem enviada pelo iOS quando o usuário interagir com o aplicativo. A primeira parte do código está abaixo:

```
1 // TapCirculoView.m
2 #import "TapCirculoView.h"
3 
4 @interface TapCirculoView()
5 -(void) registraEventos;
6 @end
7 
```

```
8 @implementation TapCirculoView
9
10 -(id) init {
11     if ((self = [super init])) {
12         [self registraEventos];
13     }
14
15     return self;
16 }
```

O código das linhas 4 a 6 apenas declara a existência do método `registraEventos` somente para o compilador não emitir um alerta, conforme vimos no capítulo sobre Objective-C. Já no método `init` (que é o construtor da classe), aproveitamos para registrar os eventos de toque, conforme abaixo:

```
1 // TapCirculoView.m
2 -(void) registraEventos {
3     UITapGestureRecognizer *toque = [[UITapGestureRecognizer alloc]
4         initWithTarget:self action:@selector(toqueRecebido:)];
5
6     toque.numberOfTapsRequired = 1; // Valor padrão é 1
7     [self addGestureRecognizer:toque];
8 }
```

Ao contrário do exemplo apresentado no início do capítulo, que sobrescrevia os métodos `touchesBegan` e `touchesEnded` da classe `UIView`, os reconhecedores de gesto são classes independentes, podendo inclusive ser criadas em qualquer outra classe. No nosso caso, isso é feito no método `registraEventos`. A classe `UITapGestureRecognizer` contém a propriedade `numberOfTapsRequired`, que serve para informar quantos toques o usuário deverá fazer para que a ação seja disparada, o que é feito na linha 6. Na linha 7 adicionamos o gesto na view, para que o iOS possa identificar as ações no momento em que for apropriado — ou seja, sem o código da linha 7, nada aconteceria.

Todos reconhecedores de gestos seguem a mesma estrutura de construção, esperando a classe que irá tratar os eventos (`initWithTarget`) e o método naquela classe (`action`) que será responsável pela ação em si. Isso é feito nas linhas 3 e 4, onde passamos a própria view como delegate, e a "referência" ao método que deverá ser executado, cuja implementação está abaixo:

```
1 // TapCirculoView.m
2 -(void) toqueRecebido:(UIGestureRecognizer *) gesto {
```

```
3       if (!circulos) {
4           circulos = [[NSMutableArray alloc] init];
5       }
6
7       CGPoint localizacaoToque = [gesto locationInView:self];
8       [circulos addObject:[NSValue valueWithCGPoint:localizacaoToque]];
9       [self setNeedsDisplay];
10  }
```

O valor do argumento `action` deve seguir um dos dois formatos abaixo:

```
- (void) gestoRecebido;
- (void) gestoRecebido:(UIGestureRecognizer *) gesto;
```

A diferença entre ambos é que o segundo recebe como argumento o reconhecedor de gesto (*tap, swipe, panning* etc.) que enviou a mensagem, o que pode ser útil tanto para utilizar o mesmo método para diversos eventos, ou — o caso mais corriqueiro — para obter algumas informações adicionais necessárias para a execução da lógica.

O método `toqueRecebido:` é o correspondente ao `touchesEnded` do exemplo anterior, não havendo diferenças substanciais na lógica, tanto que o código da linha 7 é virtualmente igual à outra implementação, porém utilizando um `UIGestureRecognizer`.

A classe `TapCirculoView` ainda precisa do método `drawRect:`, que você pode copiar integralmente do exemplo anterior. Lembre-se também de mudar o método `loadView` da classe `ViewController` para o seguinte:

```
-(void) loadView {
    TapCirculoView *c = [[TapCirculoView alloc] init];
    c.backgroundColor = [UIColor whiteColor];
    self.view = c;
}
```

8.4 Tremedeira com toque longo

Os ícones do iOS são bastante medrosos, bastando segurá-los por algum tempo para que comecem a tremer desesperadamente. Para demonstrar o gesto de toque longo (*long press*) e o uso de mais de um gesto na mesma view, vamos fazer um aplicativo que também faz tremer as views, porém dando a opção de acabar com o sofrimento

delas realizando um duplo toque (*double tap*). Além disso, este exemplo mostrará recursos mais avançados de decoração das views.

Crie um novo projeto chamado "TremeTreme" com as configurações padrão de sempre, e já adicione uma classe chamada `TremeTremeView`, que estende de `UIView`. Como o código é um pouco maior que de costume, iremos ver a implementação por partes. Todo o código a seguir é referente ao arquivo `TremeTremeView.m`, não há nenhum código no `TremeTreme.h` desta vez.

```
1  #import "TremeTremeView.h"
2  #import <QuartzCore/QuartzCore.h>
3
4  #define RADIANS(degrees) ((degrees * M_PI) / 180.0)
5
6  @interface TremeTremeView ()
7  -(void) registraGestos;
8  -(void) enfeitaView;
9  @end
10
11 @implementation TremeTremeView
12
13 -(id) initWithFrame:(CGRect)frame {
14     if ((self = [super initWithFrame:frame])) {
15         [self enfeitaView];
16         [self registraGestos];
17     }
18
19     return self;
20 }
```

Entre as linhas 1 e 9 temos algumas declarações gerais para que o resto da classe possa funcionar corretamente, sendo que a parte nova é a linha 2, que importa o framework de animação Quartz, e a linha 4 que define uma constante `RADIANS` para converter graus para radianos. Além disso, note que na linha 13 estamos declarando o inicializador `initWithFrame:` ao invés de apenas `init`, pois é o método que será utilizado mais adiante no controlador para criar instâncias desta view já especificando seu tamanho. **Muito importante:** como estamos sobrescrevendo `initWithFrame:`, na linha 14 é necessário também executar o `initWithFrame:` de `super`, caso contrário o seu aplicativo se comportará de maneiras estranhas.

Seguindo, temos o método `enfeitaView`:

```
-(void) enfeitaView {
    self.layer.masksToBounds = NO;
    self.layer.cornerRadius = 8;
    self.layer.shadowOffset = CGSizeMake(-2, 2);
    self.layer.shadowRadius = 5;
    self.layer.shadowOpacity = 0.5;
}
```

O código deste método depende daquele import do `QuartzCore` feito anteriormente, para que possamos acessar a propriedade `layer`. Na linha 3 dizemos que as bordas da view devem ser arredondadas em 8 pixels. Na linha 4 é adicionado uma pequena sombra com o tamanho do código existente na linha 5, enquanto que a linha 6 especifica em 50% a opacidade da sombra. Com estas opções o aspecto visual ficará muito mais agradável para o usuário (experimente remover este método, para ver a diferença causada).

Os gestos são criados no método `registraGestos`:

```
-(void) registraGestos {
    UILongPressGestureRecognizer *toqueLongo =
        [[UILongPressGestureRecognizer alloc] initWithTarget:self
                                        action:@selector(iniciaTremedeira:)];
    toqueLongo.minimumPressDuration = 0.3;
    [self addGestureRecognizer:toqueLongo];

    UITapGestureRecognizer *toqueParar =
        [[UITapGestureRecognizer alloc] initWithTarget:self
                                        action:@selector(pararAnimacao)];
    toqueParar.numberOfTapsRequired = 2;
    [self addGestureRecognizer:toqueParar];
}
```

A principal novidade é a classe `UILongPressGestureRecognizer`, utilizada para gestos quando o usuário toca a tela e mantém o dedo pressionado por um determinado tempo até disparar o evento de iniciar a tremedeira da view, que no nosso caso são 300 milissegundos, conforme feito na linha 4. Além disso, precisamos de uma maneira de fazer a view parar de tremer, e para tanto é utilizado o `UITapGestureRecognizer` com 2 toques. Resumindo: toque e segure o dedo em cima da view por pelo menos 300 milissegundos para fazê-la começar a tremer, e depois toque 2 vezes rapidamente para parar.

O código que inicia a tremedeira está abaixo:

```
-(void) iniciaTremedeira:(UIGestureRecognizer *) gesto {
    CGAffineTransform oscilaEsquerda = CGAffineTransformRotate(
        CGAffineTransformIdentity, RADIANS(-5.0));

    CGAffineTransform oscilaDireita = CGAffineTransformRotate(
        CGAffineTransformIdentity, RADIANS(5.0));

    self.transform = oscilaEsquerda;

    UIViewAnimationOptions opcoes = UIViewAnimationOptionRepeat
        | UIViewAnimationOptionAllowUserInteraction
        | UIViewAnimationOptionAutoreverse
        | UIViewAnimationCurveEaseInOut;

    [UIView animateWithDuration:0.1 delay:0 options:opcoes animations:^{
        self.transform = oscilaDireita;
    } completion:nil];
}
```

A lógica é a seguinte: queremos fazer a view girar rapidamente 5 graus para a esquerda e 5 graus para a direita, o que é conseguido com o uso de *transformações* do `Core Animation`. Nas linhas 2 e 3 definimos a oscilação para a esquerda, enquanto nas linhas 5 e 6, a oscilação para a direita, e as aplicamos na propriedade `transform` da própria view, conforme as linhas 8 e 16. Entre as linhas 15 e 17 criamos e iniciamos a animação, da seguinte maneira: cada oscilação deverá durar 100 milissegundos (`animateWithDuration:0.1`), sendo que deve começar imediatamente (`delay:0`). Para que tudo funcione corretamente, passamos uma série de opções do tipo `UIViewAnimationOptions` que são determinadas entre as linhas 10 e 13. `UIViewAnimationOptionRepeat` informa que o bloco de animação deve se repetir indefinidamente, `UIViewAnimationOptionAllowUserInteraction` é necessário para que o duplo toque de parar seja identificado, pois caso contrário o iOS irá ignorar interações com a view enquanto ela estiver sendo animada. `UIViewAnimationOptionAutoreverse` especifica que, ao chegar no final, a animação deve automaticamente voltar para o ponto inicial (que é o da linha 8), enquanto que a opção `UIViewAnimationCurveEaseInOut` é um ajuste fino na maneira como a animação ocorre.

Para parar a tremedeira, implemente o seguinte método:

```
-(void) pararAnimacao {
    UIViewAnimationOptions opcoes =
```

```
3               UIViewAnimationOptionBeginFromCurrentState;
4
5       [UIView animateWithDuration:0 delay:0 options:opcoes animations:^{
6           self.transform = CGAffineTransformIdentity;
7       }completion:nil];
8   }
```

A lógica deste código pode não ser muito clara a princípio, porém funciona assim: cada `UIView` independente somente pode ter um bloco de animação sendo executado por vez, e não existe um método "parar animação". Portanto, o que fazemos é disparar *outra* animação que dura zero segundos (`animateWithDuration:0`), que retorna a view para o estado normal, conforme a linha 5. O código da linha 2 é importante para que a transição seja feita a partir do estado em que a view se encontra no momento — em outras palavras, para ficar mais suave.

A última parte do código consiste na classe `ViewController.m`, conforme abaixo:

```
1  #import "ViewController.h"
2  #import "TremeTremeView.h"
3
4  @implementation ViewController
5
6  -(void) criaTremeTremeView:(CGPoint) posicao comCor:(UIColor *) cor {
7      CGRect r = CGRectMake(posicao.x, posicao.y, 60, 60);
8      TremeTremeView *t = [[TremeTremeView alloc] initWithFrame:r];
9      t.backgroundColor = cor;
10     [self.view addSubview:t];
11 }
12
13 - (void)viewDidLoad {
14     [super viewDidLoad];
15     self.view.backgroundColor = [UIColor whiteColor];
16
17     [self criaTremeTremeView:CGPointMake(50, 50)
18                             comCor:[UIColor blueColor]];
19     [self criaTremeTremeView:CGPointMake(200, 150)
20                             comCor:[UIColor greenColor]];
21     [self criaTremeTremeView:CGPointMake(110,250)
22                             comCor:[UIColor purpleColor]];
```

```
23 }
24 @end
```

O método `CriaTremeTremeView:comCor:` das linhas 6 a 11 é utilizado nas linhas 17, 19 e 21, sendo responsável por criar instâncias da `TremeTremeView` em um lugar centralizado — afinal, duplicar código nunca é bom, certo?!

O resultado deste aplicativo está na figura 8.2.

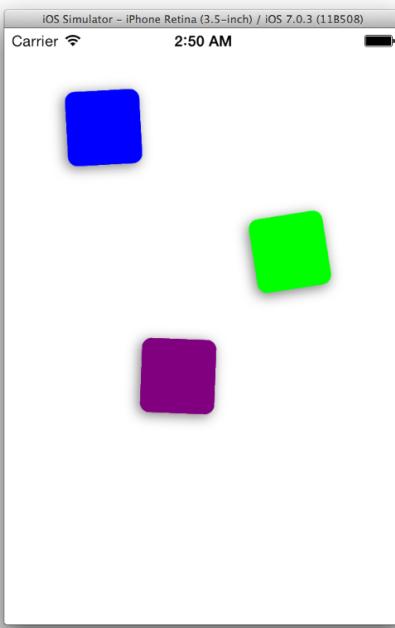



Figura 8.2: Views tremendo de medo. Repare nas habilidades visuais do autor

8.5 Deslizando uma view com o gesto Swipe

O próximo exemplo demonstra o uso do gesto *swipe*, que é um movimento que se assemelha a jogar (ao invés de arrastar) algo de uma direção para outra com o dedo, como navegar pelas imagens de uma galeria de fotos. O programa consistem em um retângulo que pode ser jogado de um lado para o outro. Crie um novo projeto chamado "GestoSwipe" e declare uma variável do tipo `UIView` chamada "quadrado" no

arquivo `ViewController.h`, conforme exemplificado abaixo. Este será o objeto que manipularemos com o gesto.

```
// ViewController.
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController {
    UIView *quadrado;
}

@end
```

Feito isso, precisamos implementar a lógica em si, no arquivo `ViewController.m`. Como os gestos devem necessariamente estar funcionando quando o aplicativo estiver pronto para o uso, precisamos registrá-los de maneira automática quando a tela for carregada, para que não haja a possibilidade de esquecermos de fazer isso manualmente em algum outro momento. O método `viewDidLoad` da classe `UIViewController` pode ser sobrescrito para fazer este trabalho, pois é executado automaticamente uma única vez, durante a inicialização do controlador. Veja o código abaixo.

```
// ViewController.m
- (void)viewDidLoad {
    [super viewDidLoad];
    [self criaQuadrado];
    [self registraGestos];
}
```

A primeira coisa a fazer é criar a view que será manipulada em cada gesto, representada pela variável `quadrado`:

```
-(void) criaQuadrado {
    quadrado = [[UIView alloc]
            initWithFrame:CGRectMake(100, 100, 100, 100)];
    quadrado.backgroundColor = [UIColor yellowColor];
    [self.view addSubview:quadrado];
}
```

Já o método `registraGestos`, responsável por adicionar os *swipes*, segue abaixo:

```
-(void) registraGestos {
    [self adicionaGesto:UISwipeGestureRecognizerDirectionRight];
    [self adicionaGesto:UISwipeGestureRecognizerDirectionLeft];
}

-(void) adicionaGesto:(UISwipeGestureRecognizerDirection) direcao {
    UISwipeGestureRecognizer *swipe = [[UISwipeGestureRecognizer alloc]
        initWithTarget:self action:@selector(jogaQuadrado:)];
    swipe.direction = direcao;
    [self.view addGestureRecognizer:swipe];
}
```

É no método `adicionaGesto:` que o trabalho de adicionar os gestos é feito de fato. Note que precisamos fazer isso para cada direção desejada, que é especificada na propriedade `direction`, na linha 9. Já o método que lida com o *swipe* precisa primeiro verificar se o usuário fez o movimento dentro da área do retângulo, e então deslocar o mesmo para a esquerda ou para a direita:

```
-(void) jogaQuadrado:(UIGestureRecognizer *) gesto {
    CGPoint location = [gesto locationInView:quadrado];

    if ([quadrado pointInside:location withEvent:nil]) {
        UISwipeGestureRecognizer *swipe =
            (UISwipeGestureRecognizer *)gesto;

        float novoX = swipe.direction ==
            UISwipeGestureRecognizerDirectionLeft ? 0
            : (self.view.frame.size.width - quadrado.frame.size.width);

        [UIView animateWithDuration:0.3 delay:0
            options:UIViewAnimationOptionCurveEaseInOut
            animations:^ {
                CGRect frame = quadrado.frame;
                frame.origin.x = novoX;
                quadrado.frame = frame;
            } completion:nil];
    }
}
```

Na linha 4 verificamos se o toque ocorreu na área do quadrado propriamente dito, e caso isso seja verdade, definimos a nova posição horizontal do quadrado de

acordo com a direção do *swipe*, nas linhas 7 a 9. O restante do método simplesmente anima a troca de posição para a posição do `novoX`.

Rode o application com `Command + R` e faça os movimentos do gesto (se for com o mouse, clique e arraste para simular o mesmo efeito do dedo). A view `quadrado` deverá deslizar para a direção determinada.

Capítulo 9

Trabalhe com mapas e GPS na sua aplicação

Este é sem dúvida um recurso que aparece em muitas aplicações, às vezes em contextos totalmente diferentes do que esperamos. A API não é complicada e você verá que com o conhecimento que obteve até aqui é fácil atacar uma nova biblioteca. Até a versão 5 do iOS o sistema de mapas utilizado é o do Google, enquanto que a partir do iOS 6 os mapas são da Apple.

O código-fonte deste capítulo está disponível na pasta "MinhaLocalizacao" (lembrando que o endereço do site com os códigos está na introdução do livro).

9.1 As bibliotecas necessárias

Crie um novo projeto com o sugestivo nome de `MinhaLocalizacao`, utilizando as mesmas configurações padrão dos outros projetos feitos até agora no livro.

A primeira coisa a fazer é adicionar as bibliotecas necessárias para trabalhar com

mapas, chamada de `MapKit`, e a de localização, chamada `CoreLocation`. Abra as propriedades do projeto, selecione a aba *Build Phases*, e expanda a seção *Link Binary with Libraries*, conforme mostra a figura 9.1. Clique no botão + e adicione o item `MapKit.framework` e `CoreLocation.framework`.

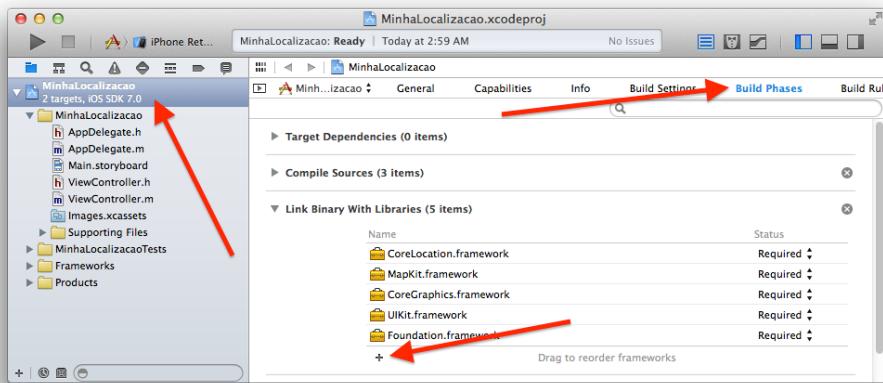



Figura 9.1: Adicionando as bibliotecas MapKit e CoreLocation

9.2 Adicionando o mapa à aplicação

Adicionar mapas aos nossos aplicativos é uma tarefa surpreendentemente simples, pois o SDK do iOS já vem com um componente pronto com virtualmente tudo o que é necessário para ter o mapa funcionando. Não é necessário nos preocuparmos com questões de rede, posicionamento de imagens, performance e afins, o que nos deixa livre para concentrar esforço na lógica de negócios principal.

Abra o arquivo `Main.storyboard` e adicione o componente `Map View`, localizado na *Object Library* (`CTRL + Option + Command + 3`, ou pelo menu *View -> Utilities -> Show Object Library*), posicionando-o de tal forma que ocupe toda a área visível do aplicativo. Além disso vamos precisar manipular o mapa diretamente pelo código, portanto é necessário criar um `Outlet` para ele: abra o *Assistant Editor* (`Option + Command + ENTER`), selecione o mapa, segure o `CTRL` e arraste para o arquivo `ViewController.h`, dando o sugestivo nome "mapa".

Se você tentar compilar o aplicativo verá que o Xcode reclama que não sabe o que significa o tipo `MKMapView` - isso deve-se ao fato que ele está localizado

em um arquivo de cabeçalho específico, `MapKit/MapKit.h`. Altere o arquivo `ViewController.h` conforme o código abaixo. Repare que na linha 2 importamos o cabeçalho correto do `MapKit`.

```
1 #import <UIKit/UIKit.h>
2 #import <MapKit/MapKit.h>
3
4 @interface ViewController : UIViewController
5 @property (retain, nonatomic) IBOutlet MKMapView *mapa;
6
7 @end
```

Rode o aplicativo e interaja com o mapa: arraste-o, dê duplo clique para realizar zoom, e veja que tudo funciona muito bem!

9.3 Simulando múltiplos toques

Diversos aplicativos fazem uso de múltiplos toques para realizar certas ações. No caso de mapas, toque duplo com um único dedo (duplo clique do mouse no simulador) aumenta o zoom, enquanto que um toque simples com *dois* dedos diminui o zoom. Para simular toque com dois dedos basta segurar a tecla `Option` e fazer o movimento de *pinch in* e *pinch out* com o mouse. A figura 9.2 mostra o aplicativo rodando e o indicador visual de toques múltiplos.

Figura 9.2: Indicador visual de toques múltiplos

9.4 Posicionando o mapa automaticamente na localização do usuário

Para mostrar a localização atual do usuário, precisamos apenas definir a propriedade `showsUserLocation` para `YES`:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.mapa.showsUserLocation = YES;
}
```

Rode o aplicativo novamente e ele deverá mostrar uma mensagem solicitando permissão para acessar a sua localização atual, conforme a figura 9.3. Clique em "OK" para permitir.

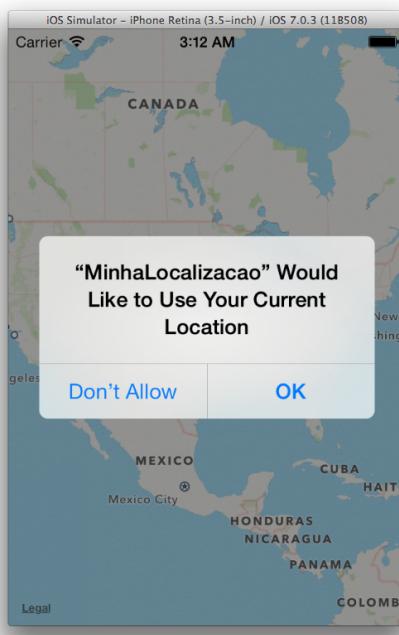



Figura 9.3: Aplicativo solicitando acesso à localização do usuário

Para facilitar os testes, o iOS Simulator permite a inserção manual de qualquer latitude e longitude, conforme mostra a figura 9.4. Basta acessar o menu *Debug -> Location -> Custom Location...*

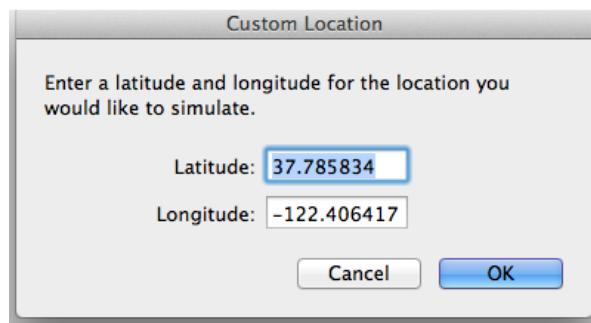



Figura 9.4: Inserindo manualmente uma determinada latitude e longitude

9.5 Trabalhe com o zoom

Embora o aplicativo mostre a nossa posição, a visão do mapa está sendo vista muito de cima, e o que queremos é apresentar ao usuário quase ao nível da rua. Para definir o nível de zoom é necessário especificar o tamanho desejado da área visível, o qual é representado em um sistema de coordenadas próprio do `MapKit`. Existem diversas maneiras e momentos em que é possível realizar esta ação, e a que apresentaremos como exemplo consiste em modificar o zoom no momento em que o mapa recebe aquele pininho azul que aponta a localização do usuário.

O componente `Map View` tem diversos eventos os quais podemos escutar, e um deles diz respeito ao momento em que algum pino é adicionado ao mapa — o `mapView:didAddAnnotationViews:`, definido no delegate `MKMapViewDelegate`, sendo necessário importá-lo no arquivo `ViewController.h` conforme o código abaixo:

```
@interface ViewController : UIViewController<MKMapViewDelegate>
@property (weak, nonatomic) IBOutlet MKMapView *mapa;
@end
```

Falta informar ao `mapa` onde estarão implementados os métodos do delegate, no caso, a própria `ViewController.m`:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.mapa.showsUserLocation = YES;

    // Define a própria classe ViewController como tendo os métodos
    // do delegate do MapKit
    self.mapa.delegate = self;
}
```

E a implementação do método `mapView:didAddAnnotationViews:` :

```
1 -(void) mapView:(MKMapView *)mapView
2             didAddAnnotationViews:(NSArray *)views {
3     MKAnnotationView *v = [views objectAtIndex:0];
4     CLLocationDistance distancia = 400;
5     MKCoordinateRegion regiao = MKCoordinateRegionMakeWithDistance(
6         [v.annotation coordinate], distancia, distancia);
7     [self.mapa setRegion:regiao animated:YES];
8 }
```

As linhas 4 e 5 definem a região do zoom com base na distância em metros definida na linha 3. Rode-o novamente, e o resultado deverá ser algo como a figura 9.5.

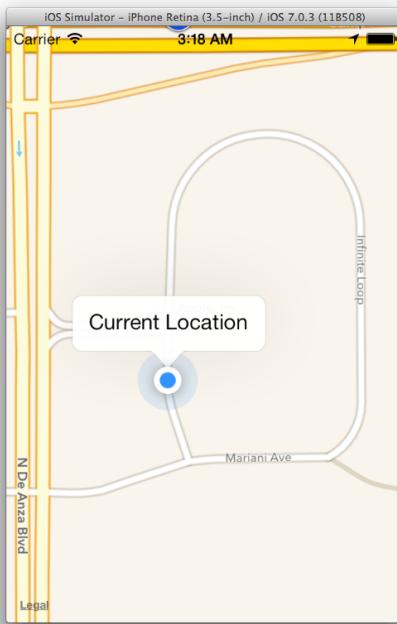



Figura 9.5: Mapa com zoom aplicado

9.6 Adicionando pinos ao mapa

Para finalizar o aplicativo de mapas, vamos permitir que o usuário insira pinos em qualquer lugar ao realizar um toque longo de pelo menos 500 milissegundos, e centralizar o mapa na região deste pino. A primeira coisa a fazer é adicionar ao mapa um gesto do tipo `UILongPressGestureRecognizer`, conforme o código abaixo:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.mapa.showsUserLocation = YES;
    self.mapa.delegate = self;
```

```
    // Adiciona ao mapa o gesto de toque longo
    UILongPressGestureRecognizer *toqueLongoMapa =
        [[UILongPressGestureRecognizer alloc] initWithTarget:self
            action:@selector(adicionaPino:)];

    toqueLongoMapa.minimumPressDuration = 0.5;
    [self.mapa addGestureRecognizer:toqueLongoMapa];
}
```

A implementação do método `adicionaPino:` segue abaixo:

```
1  -(void) adicionaPino:(UIGestureRecognizer *) gesto {
2      if (gesto.state == UIGestureRecognizerStateBegan) {
3          CGPoint ponto = [gesto locationInView:self.view];
4
5          CLLocationCoordinate2D coordenadas =
6          [self.mapa convertPoint:ponto toCoordinateFromView:self.mapa];
7
8          MKPointAnnotation *pino = [[MKPointAnnotation alloc] init];
9          pino.coordinate = coordenadas;
10         [self.mapa addAnnotation:pino];
11     }
12 }
```

A verificação na linha 2 é necessária para que o pino seja adicionado apenas uma única vez, dado que o gesto de toque longo dispara o evento quando o dedo é retirado da tela também. Nas linhas 5 e 6 convertemos a posição do toque na tela para o sistema de coordenadas do mapa. O resto do método simplesmente cria o pino e adiciona-o ao mapa.

Ao realizar um toque longo, o método `mapView:didAddAnnotationViews:` implementado anteriormente será automaticamente executado pelo iOS, centralizando o pino na tela, conforme mostra a figura 9.6.

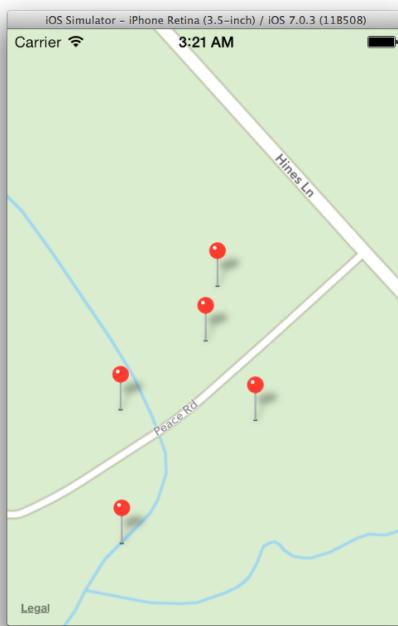



Figura 9.6: Resultado do aplicativo após a adição de diversos pinos

9.7 Detecte toques nos pinos

O delegate `MKMapViewDelegate` tem diversos métodos interessantes, como o `mapView:didSelectAnnotationView:`, que é executado toda vez que o usuário interage com algum pino. Veja um exemplo no código abaixo, onde apenas mostramos uma mensagem de log quando isso ocorre — um uso mais criativo fica como lição de casa para você!

```
- (void)mapView:(MKMapView *)mapView
        didSelectAnnotationView:(MKAnnotationView *) pino {
    NSLog(@"Pino %@ selecionado", pino);
}
```

Capítulo 10

Componentes gráficos customizados

Com raras exceções, todos os componentes gráficos no iOS — botões, seletores de datas e valores, campos de texto, imagens, barras de navegação e tantos outros — descendem da classe `UIView`, que é também a classe que devemos estender para criar nossos próprios componentes visuais customizados. Na prática ela é relativamente simples, definindo uma área retangular na tela e as interfaces para gerenciar conteúdo naquela área, tais como desenho e animação, eventos, layout e outras views-filhas.

Neste capítulo veremos as principais propriedades e estrutura da `UIView`, de tal forma que seja possível aplicar o conhecimento na construção de aplicativos ricos e interativos, assim como criar seus próprios componentes. Ao contrário do resto do livro, aqui o conteúdo é apresentado num estilo mais próximo de guia de referência, inclusive por sua simplicidade.

Views podem ser criadas totalmente na mão, com código puro, ou com um mix de código e layout visual, feito em conjunto com arquivos `.xib` através do Interface Builder. O resultado prático é idêntico, sendo que arquivos XIB são úteis para

elementos estáticos - caso queira manipular o que aparece e quando aparece, será necessário fazer via código a parte dinâmica.

Geralmente a melhor alternativa é criar a tela em arquivos XIB, e fazer via código o que for dinâmico e / ou condicional.

Desenho e animação

- O conteúdo é desenhado utilizando tecnologias como `UIKit`, `Core Graphics` e `OpenGL ES`. - Certas propriedades da view podem ser modificadas utilizando efeitos de animação. Veremos como fazer isso neste capítulo.

Gerenciamento de layout e views-filhas

- Uma `UIView` pode opcionalmente conter diversas outras views-filhas, chamadas de `subviews`. - Cada view pode definir as regras de redimensionamento em relação à view-pai. - Uma view-pai pode modificar o tamanho das views-filhas.

Gerenciamento de eventos

- Toda view pode lidar com eventos de toque e outros tipos de interação feitos pelo usuário.

10.1 Criando views

Por ser retangular, para criar views é necessário especificar as coordenadas x e y, além da largura e altura, conforme o exemplo abaixo:

```
1 CGRect retangulo = CGRectMake(20, 20, 200, 200);
2 UIView quadradoView = [[UIView alloc] initWithFrame:retangulo];
```

A API de `UIView` utiliza o nome `frame` (quadro, moldura) para se referenciar às coordenadas e dimensões da view. Você trabalhará bastante com esta propriedade. Para modificar o frame é necessário passar um `CGRect` completo novamente, como no exemplo abaixo:

```
retangulo.frame = CGRectMake(50, 100, 150, 300);
```

Tecnicamente o `frame` é representado por uma `struct` chamada `CGRect`, que por sua vez contêm outras duas `struct`s chamadas `size` (que é do tipo `CGSize`)

e `origin` (do tipo `CGPoint`), que correspondes ao tamanho e posição respectivamente. Isso significa que para *pegar* o tamanho ou posição de qualquer view, basta fazer como no exemplo abaixo:

```
float x = retangulo.frame.origin.x;
float y = retangulo.frame.origin.y;
float largura = retangulo.frame.size.width;
float altura = retangulo.frame.size.height;
```

Por outro lado, para *modificar* apenas algumas uma das propriedades (`x`, `y`, `width` ou `height`) é necessário um pouco mais de trabalho, pois é preciso primeiro atribuir o `frame`% a uma variável temporária, modificá-la e atribuir de volta ao `frame`. Veja o exemplo abaixo:

```
1 CGRect frame = retangulo.frame;
2 frame.origin.x = 350;
3 retangulo.frame = frame;
```

Escrever apenas `retangulo.frame.origin.x = 350` não funcionaria, pois como estamos lidando com structs em C, e frame é um valor (não um ponteiro) de UIView, tentar alterá-lo dessa forma não surtiria efeito (alteraria a cópia do frame em memória), e o compilador não permitiria. É um código bastante repetitivo, e no capítulo sobre Objective-C avançado veremos uma forma de tornar essa tarefa mais simples.

10.2 Animando views

Além de darem um aspecto mais bonito para a sua aplicação, animações são maneiras muito úteis de informar ao usuário que alguma coisa está acontecendo. A própria Apple recomenda que o uso deste recurso seja sempre considerado. Diversas propriedades da `UIView` têm suporte à animação de maneira extremamente fácil, bastando apenas informar que desejamos realizá-la, e modificar a propriedade em si. As seguintes propriedades podem ser animadas desta forma:

- `frame`: para animar tamanho e posição
- `bounds`: tamanho e posição
- `center`: animar a posição

- `transform`: para rotacionar ou fazer scale
- `alpha`: para animar a mudança de opacidade
- `contentStretch`: para modicar a maneira como a view se estica

Um lugar onde animações são bastante úteis é em transição de telas ou propriedades — sem contar, é claro, os aspectos puramente estéticos. Existem duas maneiras de animar estas propriedades: uma procedural, utilizando a dupla `beginAnimation` e `commitAnimation`, e outra mais "OO" com o uso de blocos. O resultado é exatamente o mesmo independentemente da técnica, contudo a partir do iOS 4 a maneira mais utilizada é a com blocos.

10.3 Animando da forma procedural e tradicional

A maneira mais tradicional de animar propriedades dos componentes é bastante procedural e simples de entender e escrever. A técnica consiste em *marcar* o início daquilo que desejamos animar, e depois *aplicar* a animação propriamente dita.

O código abaixo demonstra isso:

```
1 [UIView beginAnimations:nil context:nil];
2 [UIView setAnimationDuration:2];
3
4 minhaView.frame = CGRectMake(20, 308, 280, 132);
5 minhaView.alpha = 0.3;
6
7 [UIView commitAnimations];
```

Na linha 1 preparamos o ambiente, indicando que o código que vem abaixo deverá ser executado em um bloco de animação, enquanto que o na linha 7 informamos o fim desta instrução — tudo o que vem no meio será animado. O código na linha 2 especifica a duração da animação, sendo que o valor padrão é em torno de 300 milissegundos. Já nas linhas 4 e 5 é feito o código que será animado de fato. O código completo está no projeto de exemplo `AnimacaoTradicionalEx1`.

10.4 Animando com o uso de blocos

Blocos é um conceito que apareceu no iOS 4 e que torna muito mais simples (do ponto de vista de exigir menos código) a execução de diversas tarefas, incluindo a de

animação de views. Para realizar a mesma operação do exemplo anterior utilizando blocos, o código ficaria conforme exemplificado abaixo:

```
[UIView animateWithDuration:2 animations:^ {
    minhaView.frame = CGRectMake(20, 308, 280, 132);
    minhaView.alpha = 0.3;
}];
```

Apesar da sintaxe um tanto estranha, o código fica muito mais enxuto.

10.5 Criando views customizadas

Views customizadas são muito úteis quando há a necessidade de reutilizar código de componentes visuais — por exemplo, seu próprio botão, componente de imagem que já apresenta uma descrição da foto, menu com suporte a eventos, desenho e virtualmente qualquer coisa que você possa imaginar. Além disso, é bem simples fazer isso: basta estender a classe `UIView`. Para melhor exemplificar, vamos criar um componente para realizar um hipotético login na aplicação, com campo de usuário, senha e o botão para validar os dados, conforme mostrado na imagem 10.1. O código do aplicativo de exemplo focará na parte de criar a tela customizada, deixando de lado qualquer lógica específica de autenticação propriamente dita.

Figura 10.1: Componente de login adicionado no aplicativo

Crie um novo projeto do tipo *Single View Application* e em seguida adicione uma nova classe chamada "LoginView" (Command + N), selecionando `UIView` no campo "Subclass of". A tela de login poderá ser posicionada em qualquer lugar, porém a altura e largura deverão ser sempre as mesmas.

Coloque tudo em LoginView.m

Todos os fragmentos de código a seguir deverão ser adicionados no arquivo `LoginView.m`.

Fragmentos para simplicidade e foco

Criar e configurar componentes visuais no iOS é uma tarefa bastante monótona e repetitiva, frequentemente acompanhada de dezenas de linhas de código cuja única função é definir propriedades de cada um dos componentes. Para manter a objetividade, aqui no livro iremos colocar apenas os pedaços principais, de tal forma que seja

possível demonstrar o que faremos porém sem poluir demais as páginas. O código completo você pode baixar no endereço http://bit.ly/ios-ViewCustomizada.

O que devemos fazer

Usando a imagem 10.1 como referência, o componente `LoginView` precisa ter o fundo com sombra que conterá toda a estrutura, uma caixa com cantos arredondados onde, dentro dela, existem dois componentes de texto para o nome de usuário e senha, e dois componentes para o usuário informar os respectivos valores. Por último, há um botão azul. Todos eles precisam ser instanciados e configurados manualmente, incluindo as dimensões e localização.

Para começar vamos sobrescrever o método `init` para que, ao ser instanciado, o componente já se autoconfigure, evitando assim que o desenvolvedor tenha que se lembrar de fazer ajustes manualmente toda vez. Confira o seguinte código:

```
-(id) init {
    if ((self = [super init])) {
        [self constroiTela];
    }

    return self;
}
```

Esta parte é bastante simples, e basicamente delega a tarefa de criar os componentes ao método `constroiTela`, cuja *primeira parte* está abaixo:

```
1 -(void) constroiTela {
2     self.frame = CGRectMake(0, 0, 300, 150);
3     self.backgroundColor = [UIColor colorWithRed:210.0/255.0
4         green:210.0/255.0 blue:210.0/255.0 alpha:1];
5
6     // Restante do método
7 }
```

Na linha 2 definimos o `frame` do componente, para que comece na posição 0 e que tenha 300 pixels de largura e 150 de altura. Nas linhas 3 e 4 definimos a cor de fundo para cinza claro, porém a sintaxe é peculiar: ao invés da classe `UIColor` permitir que uma cor seja criada apenas com os valores RGB (210 para os três neste caso), é necessário passar o código da cor na faixa de 0.0 (preto) a 1.0 (branco), razão pela qual o código RGB é divido por 255 — maluquices da Apple.

O restante do método `constroiTela` consiste em criar cada um dos componentes restantes manualmente, definindo o `frame` de cada, opções específicas (como cor do texto ou tipo de fonte) e assim por diante — não há qualquer regra de negócios. Por este motivo, o código abaixo contêm apenas alguns fragmentos para servirem de guia para você, enquanto que o código completo pode ser baixado no endereço http://bit.ly/ios-ViewCustomizada.

```
// Borda
UIView *borda = [[UIView alloc]
                    initWithFrame:CGRectMake(10, 10, 280, 95)];
// Detalhes omitidos no livro
[self addSubview:borda];

// Labels
UILabel *usuarioLabel = [[UILabel alloc]
    initWithFrame:CGRectMake(10, 15, 100, 20)];
usuarioLabel.text = @"Usuário";
[borda addSubview:usuarioLabel];

UILabel *senhaLabel = [[UILabel alloc]
    initWithFrame:CGRectMake(10, 60, 100, 20)];
// Detalhes omitidos no livro
[borda addSubview:senhaLabel];

// Campos texto
UITextField *usuarioField = [[UITextField alloc]
    initWithFrame:CGRectMake(110, 15, 150, 20)];
// Detalhes omitidos
[borda addSubview:usuarioField];

UITextField *senhaField = [[UITextField alloc]
    initWithFrame:CGRectMake(110, 60, 150, 20)];
// Detalhes omitidos
[borda addSubview:senhaField];

// Linha horizontal divisória entre os campos
UIView *linhaFina = [[UIView alloc]
                        initWithFrame:CGRectMake(20, 50, 240, 1)];
linhaFina.backgroundColor = [UIColor greyColor];
[borda addSubview:linhaFina];
```

```
// Botao
UIButton *botaoLogin = [[UIButton alloc]
    initWithFrame:CGRectMake(0, 113, 180, 30)];
// Detalhes omitidos
[botaoLogin setTitle:@"Login" forState:UIControlStateNormal];
[self addSubview:botaoLogin];
```

Este código está resumido apenas para que você possa ter uma ideia do trabalho envolvido na adição manual de componentes à tela.

10.6 Utilizar a view customiza LoginView

Uma vez que a view tenha sido criada, basta instanciá-la como qualquer outra classe, e adicionar como `subview` em algum lugar - em um controller. por exemplo. No caso da app de demonstração "ViewCustomizada" isso é feito no método " `viewDidLoad`" da classe `ViewController.m`, conforme o seguinte código:

```
1  - (void)viewDidLoad
2  {
3      [super viewDidLoad];
4
5      LoginView *lv = [[LoginView alloc] init];
6      CGRect lvFrame = lv.frame;
7      lvFrame.origin = CGPointMake(
8          (self.view.frame.size.width - lvFrame.size.width) / 2, 30);
9      lv.frame = lvFrame;
10
11     [self.view addSubview:lv];
12 }
```

Na linha 5 o componente de login que criamos é instanciado sem qualquer diferença a tudo o que já vimos no livro, enquanto que entre as linhas 6 a 9 posicionamos em um lugar específico da tela - no caso, y 30 e centralizado no eixo horizontal. Por último, a linha 11 faz a "mágica" de adicionar o componente à tela.

10.7 Construir o componente LoginView utilizando um arquivo XIB de interface

Fazer na mão dá uma trabalheira, não?! Portanto, outra alternativa é utilizar um arquivo XIB para criar o que for possível visualmente, através do Interface Builder,

e fazer via código apenas coisas muito específicas ou dinâmicas.

Crie uma nova classe chamada "LoginViewVisual" (`CMD + N -> Objective-C Class`) que herde de "UIView" (selecione no campo "Subclass of"), e em seguida adicione um novo arquivo (`CMD + N` novamente) do tipo "View", localizado na seção "User Interface", conforme mostra a imagem 10.2. Chame o arquivo de " `LoginViewVisual.xib`", conforme a imagem 10.3.

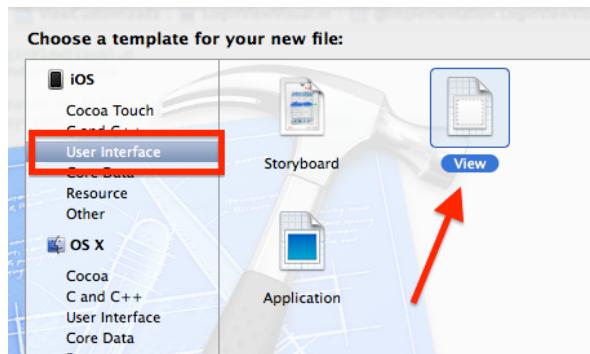



Figura 10.2: Localização do template de XIBs

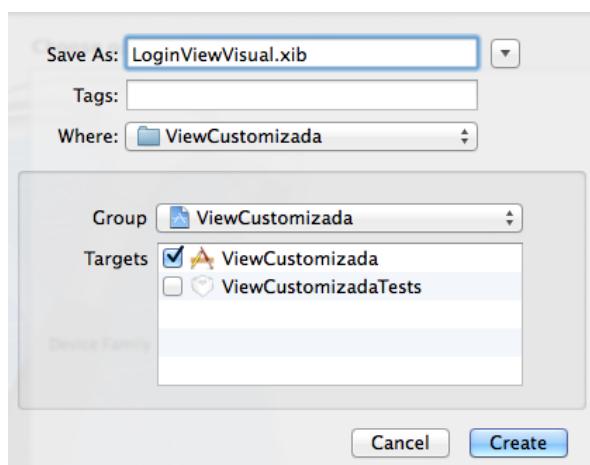



Figura 10.3: Nomeando o arquivo XIB

Atenção aos nomes

Repare que tanto o nome da classe (arquivos `.h` e `.m`) quando o arquivo de interface (`XIB`) chamam-se “ `LoginViewVisual`” - a única coisa que muda é a extensão. É muito importante prestar atenção a este detalhe, pois do contrário as coisas não funcionarão corretamente mais adiante.

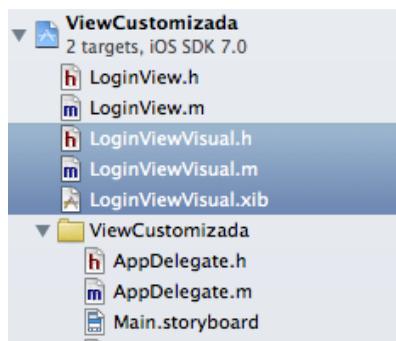



Figura 10.4: Mesmo nome, extensões diferentes

Agora abra o arquivo “ `LoginViewVisual.xib`”, e construa a tela da mesma maneira como já fizemos tantas outras vezes no livro. O tamanho do componente deve ser 300 pixels de largura, e 150 de altura. Para conseguir isso, certifique-se que o campo “Size” no Attributes Inspector esteja selecionado com a opção “Freeform”, conforme mostra a imagem 10.5.

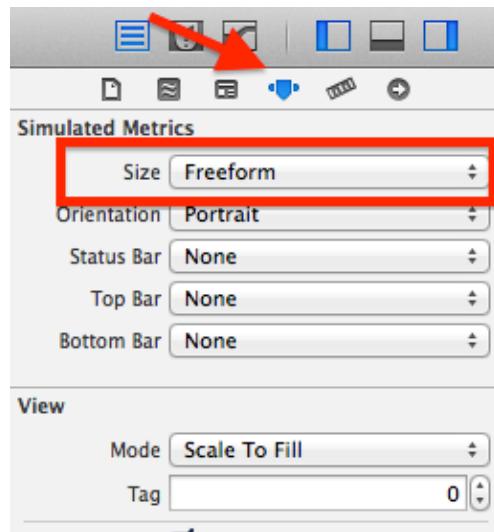



Figura 10.5: Selecione 'Freeform' para conseguir alterar o tamanho no Interface Builder

Nem tudo é mágica no mundo do iOS, e para o arquivo XIB saber que ele está associado a alguma classe, devemos informar o nome dela no campo "Custom Class" do Identity Inspector (`View -> Utilities -> Show Identity Inspector`) - no caso, "LoginViewVisual". Veja a imagem 10.6

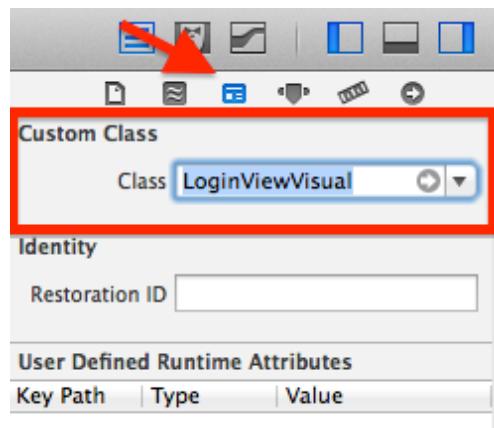



Figura 10.6: Nome da classe associada ao XIB

Crie a tela utilizando como referência a imagem 10.7 e, caso necessário, também o código anterior que fizemos para o arquivo "LoginView.m".

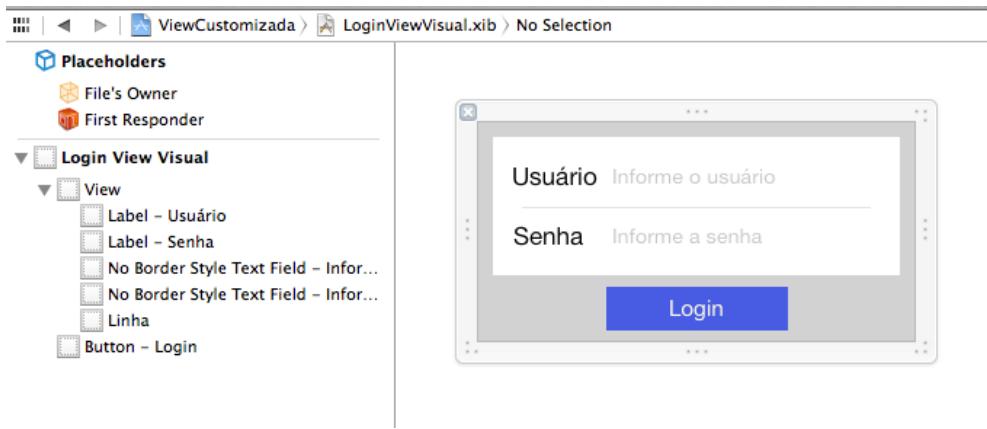



Figura 10.7: Construção da tela no Interface Builder

Conecte também os Outlets

Arquivos XIB, assim como Storyboards, representam apenas componentes visuais, nada mais. Para que seja possível interagir com eles via código é preciso conectar todos os outlets e actions que forem necessários. No caso da tela de login, crie outlets para os campos de login e senha. O seu arquivo `.h` deverá ficar assim:

```
#import <UIKit/UIKit.h>

@interface LoginViewVisual : UIView
@property (weak, nonatomic) IBOutlet UITextField *userField;
@property (weak, nonatomic) IBOutlet UITextField *passwordField;
@end
```

10.8 Como utilizar views criados com arquivos XIB

Ao contrário de componentes criados puramente com código, aqueles que tenham arquivos XIB associados precisam ser obtidos de uma outra maneira,

através da classe `NSBundle`. Confira o código abaixo, referente ao arquivo `ViewController.m`:

```
1  // ViewController.m
2  - (void)viewDidLoad
3  {
4      [super viewDidLoad];
5
6      LoginViewVisual *lv = [[NSBundle mainBundle]
7          loadNibNamed:@"LoginViewVisual" owner:self options:nil][0];
8
9      CGRect lvFrame = lv.frame;
10     lvFrame.origin = CGPointMake(
11         (self.view.frame.size.width - lvFrame.size.width) / 2, 30);
12     lv.frame = lvFrame;
13
14     [self.view addSubview:lv];
15 }
```

A diferença está nas linhas 6 e 7 onde, ao invés de instanciar a classe utilizando a dobradinha "alloc / init", obtemos através do método `loadNibNamed`, da classe `NSBundle`. O resto do código é igual ao exemplo anterior. Rode o exemplo e veja o resultado.

Não está idêntico?

Você deve ter reparado que a tela não é exatamente igual à anterior - em especial, as bordas não estão arredondadas e o botão não tem o contorno preto. Esses detalhes somente podem ser feitos via código, e não pelo Interface Builder. Fica aqui, portanto, este desafio: utilizando os conhecimentos adquiridos até agora, faça o resultado visual do componente "LoginViewVisual" ser idêntico àquele obtido com "LoginView".

Capítulo 11

Conceitos fundamentais de Objective-C

Objective-C é, no fundo, C, mas C *com esteroides*. Mais precisamente, Objective-C adiciona um conjunto de extensões à linguagem C, como orientação a objetos. A sintaxe é particularmente diferente, mas tudo o que funciona em C irá funcionar em Objective-C, inclusive bibliotecas de terceiros que nem sabem da existência da "nova" linguagem. É também a principal forma de criar aplicativos para Mac OS X e iOS, tendo sido feita pela Apple em 1983.

Aqui você pode revisar alguns conceitos, já que o enfoque do livro foi bastante prático.

11.1 Uma pequena história

Dependendo de como você encarar o aprendizado, Objective-C pode ser fácil e agradável, ou extremamente irritante e frustrante. Fácil e agradável é uma opção melhor,

mas eis uma pequena história: quando eu tive que aprender Objective-C para poder desenvolver aplicativos para iOS, eu nunca tinha visto a linguagem na frente, com exceção de algumas lidas rápidas em artigos online. A pressão para começar a criar aplicativos era imensa, precisávamos mostrar ao mercado que tínhamos um produto para atender as demandas. Isso foi logo após o lançamento mundial do iPad, e até então encontrar desenvolvedores qualificados — ou ao menos interessados — em Objective-C e desenvolvimento para iOS no Brasil era algo extremamente difícil. Portanto, durante 6 dias seguidos, das 9 da manhã às 01 da manhã do dia seguinte, minha única tarefa durante o dia todo era estudar, tentando assimilar a avalanche de informação.

Esta imersão de choque acabou ditando o desenrolar das próximas semanas. Olhando para trás, um dos maiores erros que fiz foi ter criado expectativas erradas sobre como Objective-C deveria funcionar, especialmente em relação à sintaxe. Eu queria que a sintaxe fosse como Java, C#, C, Javascript, qualquer coisa assim, e demorei para aceitar que deveria encarar a linguagem como diferente. Muitos artigos e livros descrevem Objective-C como sendo uma linguagem fácil de ler e de aprender (e talvez por isso eu tenha tido expectativas irrealistas), mas na prática não é exatamente assim.

Portanto, o meu conselho de ouro para quem estiver aprendendo Objective-C: aceite a linguagem do jeito que ela é, e tente escrever o código da maneira mais organizada que puder. A sintaxe é estranha e muitas vezes temos que dar mais voltas que o razoável. Mas é também uma linguagem poderosa, que, quando compreendida sua maneira de funcionar, torna-se agradável de trabalhar. Procure maneiras eficientes e produtivas de utilizá-la, mas não se revolte contra ela.

11.2 Nome e assinatura do método

Estes dois termos são fáceis de confundir e, grande parte do tempo, é comum os desenvolvedores referenciarem-se a eles como sendo a mesma coisa, porém existe uma diferença fundamental: o *nome* do método é composto pelas suas partes sem quaisquer referências aos tipos dos dados, enquanto a assinatura compreende o conjunto completo. Exemplo:

Assinatura: `-(void) criaEmpresaComNome:(NSString *) nome eNumeroFuncionarios:(int) quantidade`

Nome: `criaEmpresaComNome:eNumeroFuncionarios:`

Métodos em Objective-C são um dos principais pontos de confusão e reclamação

por muitos usuários, principalmente nos primeiros contatos com a linguagem. Eles são feios, verbosos e parecem repetir partes, mas depois que pegamos o jeito é até possível ver uma certa elegância. As intenções expressas no código acabam ficando mais claras, evitando idas frequentes à documentação apenas para saber a que se refere cada um dos argumentos.

Procure criar métodos semanticamente bons, que expressem de maneira clara a intenção e que sejam auto descritivos. Muitas vezes isso irá resultar em nomes mais longos, mas essa é uma prática comum, feita por todas bibliotecas. É importante utilizar a convenção já existente no ecossistema Objective-C.

Além disso, embora tenhamos usado a palavra *método* em todo o livro, o termo correto em Objective-C é "mensagem" (*message*), pois a implementação de Orientação a Objetos da linguagem é baseada no conceito de "envio de mensagens". Desta forma, nós não "invocamos um método", mas sim "enviamos uma mensagem" a um determinado objeto. Este é um conceito importante, e embora na maior parte do tempo não faça diferença *como* este mecanismo funciona, é importante entender, mesmo que superficialmente, o que ocorre por trás dos panos.

Diferente de muitas outras linguagens, em Objective-C o *destino* ("target") da *mensagem* é decidido em tempo de execução, com o objeto interpretando a mensagem enviada. A principal consequência disso é que este sistema de envio de mensagens *não* faz verificação de tipos, e na prática podemos enviar qualquer *mensagem* a qualquer *destino* (ou, em outras palavras, podemos invocar qualquer método em qualquer objeto). Contudo, isso não significa que irá funcionar, pois caso um objeto receba uma mensagem da qual não tem conhecimento, um erro em tempo de execução será gerado, muito provavelmente fazendo o aplicativo quebrar. Essa característica não deve ser encarada como negativa, muito pelo contrário, pois é um dos pilares que torna a linguagem dinâmica e permite desenvolver de formas que não são possíveis em outras linguagens.

Contudo, durante todo o livro usamos a expressão "método" e "invocar" (como em "... invocar o método `criaUsuario`"), ao invés dos termos *semanticamente* corretos segundo a especificação da linguagem. Eles são mais amigáveis ao público alvo, e acabam invariavelmente sendo utilizados pela maioria dos blogs e livros.

11.3 Propriedades

Um dos princípios de programação Orientada da Objetos é o encapsulamento, em que os atributos de uma classe nunca são acessados diretamente, mas sim através de

métodos. Muitas vezes surgem métodos de acesso e modificação — popularmente conhecidos como *getters* e *setters* —, o que permite controle sobre o gerenciamento de estado dos dados, além de expor apenas o que realmente é necessário.

Isso é facilitado em Objective-C com o uso de *propriedades*. Propriedades provêm uma sintaxe simples e limpa para métodos de acesso, além de tornar fácil a definição das regras de gerenciamento de memória e controle de concorrência múltipla.

A declaração de propriedades é feita juntamente com os demais métodos da classe, protocolo ou categoria (estes dois últimos são abordados mais adiante neste capítulo), e tem uma sintaxe própria. Considere o exemplo abaixo, retirado da classe `Empresa` que já usamos neste livro:

```
@property (nonatomic, retain) NSString *nome;
```

A diretiva `@property` declara a propriedade, enquanto que os valores entre parêntese — `(nonatomic, retain)` — são opcionais (porém geralmente utilizados) e provêm detalhes adicionais sobre o comportamento da propriedade. Utilizar uma propriedade é equivalente a declarar os métodos de acesso manualmente. Dessa forma, o código fica assim:

```
@property NSString *nome;
```

é equivalente ao código:

```
-(NSString *) nome;
-(void) setNome:(NSString *) novoNome;
```

A parte entre parênteses na declaração da propriedade pode ter os seguintes valores, que podem ser combinados com vírgula:

Acesso `readwrite` informa que a propriedade pode ser utilizada tanto para leitura (getter) quanto para escrita (setter). Este é o valor padrão.

`readonly` marca a propriedade como sendo apenas para leitura, ou seja, não será possível atribuir valores.

Atribuição `assign` informa que uma atribuição simples será feita. Se utilizado em objetos, esta opção implica que, caso o valor original seja liberado da memória, a propriedade passará a referenciar "lixo". Esta opção deve necessariamente ser usada caso a propriedade seja um tipo primitivo (como `int` e `float`) ou derivado de alguma `struct`, como `CGRect`.

retain determina que uma mensagem retain seja enviada ao objeto na hora de atribuir o valor. Desse modo, o objeto passa a participar do gerenciamento de memória. Caso a propriedade tenha algum outro valor na hora da atribuição, uma mensagem release será enviada a ele. Esse funcionamento é especialmente útil, pois evita que tenham de lidar com essas questões explicitamente.

copy é um atributo de uso mais restrito, no qual a atribuição é feita através do envio da mensagem copy, sendo ainda válido apenas para tipos que implementem o protocolo NSCopying. O valor anterior da propriedade recebe a mensagem release antes do novo valor ser atribuído.

Acesso concorrente Por padrão, todo acesso à propriedade é feito de maneira atômica, com proteção de acesso concorrente. Na prática isso significa que apenas uma thread por vez terá acesso a ela. Caso o seu programa não tenha essa necessidade (o que é 99% dos casos), utilize o atributo nonatomic.

Uma coisa boa para nós é que com propriedades não precisamos necessariamente declarar variáveis de instância. Contudo, você pode estar se perguntando: "*mas sendo assim, propriedades não acabam sendo exatamente a mesma coisa que variáveis de instância públicas?*". Esta é uma observação bastante interessante, e para entender o porquê de não ser a mesma coisa, veja os exemplos a seguir.

11.4 Acesso somente leitura

O acesso às propriedades pode ser configurado para não permitir mudanças no valor, pelo menos não para qualquer código que não seja o da própria classe:

```
1  // Empresa.h
2  @interface Teste : NSObject
3  @property (nonatomic, readonly) double lucroAnual;
4  @end
5
6  // Empresa.m
7  @implementation Empresa
8
9  -(id) init {
10     if ((self = [super init])) {
11         // Repare no uso de underline ("_")
12         _lucroAnual = 100;
13     }
14
```

```
15      return self;
16  }
17
18  -(void) calculaLucroAnual {
19      // Internamente podemos modificar a variável
20      // sem qualquer tipo de restrição
21      _lucroAnual = 3500;
22  }
23  @end
24
25  // OutraClasse.m
26  Empresa *e = [[Empresa alloc] init];
27
28  // OK, estamos lendo apenas. Imprime "100"
29  NSLog(@"Lucro atual: %f", e.lucroAnual);
30
31  // Erro, "lucroAnual" é "readonly"
32  e.lucroAnual = 700;
33
34  [e calculaLucroAnual];
35
36  // Imprime "3500"
37  NSLog(@"Lucro recalculado: %f", e.lucroAnual);
```

Repare que nas linhas 12 e 21 foi utilizado um underline ("_") como prefixo da propriedade `lucroAnual`. Isso é possível devido a uma funcionalidade do compilador chamada "*Propriedades automáticas*" (do inglês "*Auto Properties*"), one ele automaticamente "declara" a variável com underline para poder ser utilizada *dentro da classe onde foi definida*. Você pode ainda utilizar " `self.lucroAnual`" nas linhas 12 e 21, porém nas linhas 29, 32 e 37 não pode usar o prefixo com underline, pois aquele é um código externo a classe "Empresa", onde a property foi declarada.

Código customizado para o getter ou setter

Caso haja necessidade, você pode sobrescrever a implementação do método getter ou setter (ou ambos). Imagine que um `Chuveiro` tenha limites de `temperatura` que não possam ser ultrapassados:

```
// Chuveiro.h
@interface Chuveiro : NSObject
@property (nonatomic, assign) int temperatura;
```

```
@end

// Chuveiro.m
@synthesize temperatura;

-(void) setTemperatura:(int) valor {
    // Evita que o chuveiro congele ou queime a pessoa
    if (valor < 10 || valor > 30) {
        @throw [NSException exceptionWithName:@"Argumento inválido"
            reason:@"A temperatura deve ser entre 10 e 30 graus"
            userInfo:nil];

    }

    // Repare no underline
    _temperatura = valor;
}
```

A regra de nomenclatura é **set**`NomeDaPropriedade`, tomando o cuidado de que o primeiro caractere do nome da propriedade neste caso tem que ser em caixa alta. Fazendo dessa forma, o Objective-C sabe que deverá utilizar o seu método `setPosicao:`.

11.5 Utilizando propriedades dentro da própria classe

Embora possa parecer contraintuitivo, existe uma diferença singular — mas importante — na maneira como acessamos propriedades *dentro*: da classe onde foram definidas, através da palavra-chave `self`:

```
1  // Chuveiro.h
2  @interface Teste : NSObject
3  @property (nonatomic, assign) int temperatura;
4
5  -(void) acessaViaPropriedade;
6  -(void) acessaDireto;
7
8  @end
9
10 // Chuveiro.m
```

```
11 @implementation Teste
12 
13 -(void) acessaViaPropriedade {
14     self.temperatura = 15;
15 }
16 
17 -(void) acessaDireto {
18     _temperatura = 99;
19 }
20 
21 -(void) setTemperatura:(int) novaTemperatura {
22     NSLog(@"Mudando a temperatura de %d para %d", _temperatura,
23                                                novaTemperatura);
24     _temperatura = novaTemperatura;
25 }
26 @end
```

Na linha 14 estamos acessando a propriedade de fato, através de `self`. Desta forma, todas as regras definidas na declaração da propriedade na linha 3 são aplicadas — por exemplo, se tivéssemos definido-a como `readonly`, não seria possível fazer `self.temperatura = 15`. Na linha 21 sobrescrevemos o método setter da propriedade para adicionar logs. Na linha 18 não usamos `self`, o que na prática acaba funcionando como se `_temperatura` neste caso fosse uma variável de instância como qualquer outra. Repare que, como mencionado anteriormente, o compilador automaticamente cria variáveis com o nome da propriedade prefixadas com underline, que podem ser acessadas dentro da classe onde foram declaradas.

```
// OutraClasse.m
Chuveiro *c = [[Chuveiro alloc] init];

[c acessaViaPropriedade];
[c acessaDireto];
NSLog(@"Temperatura atual: %d", c.temperatura);
```

O resultado deste programa é:

```
Mudando a temperatura de 0 para 15
Temperatura atual: 99
```

11.6 Definindo protocolos

Protocolos são para Objective-C o que interfaces são para Java e C#: eles determinam um contrato que as classes que fizerem uso deles seguem. Protocolos permitem a definição de métodos opcionais, ao contrário de Java e C#, onde *todos* os métodos devem ser implementados. Isso significa que, na prática, um Protocolo representa um contrato que pode ter cláusulas opcionais, e portanto é necessário tomar algumas precauções na hora de lidar com eles. Chamamos este tipo de *protocolos informais*, em contraste aos *protocolos formais*, que são aqueles de implementação obrigatória.

Protocolos também permitem aplicar o conceito de herança múltipla em Objective-C, pois é possível que uma mesma classe atenda a diversos protocolos de uma única vez, enquanto que só pode herdar de uma única classe.

Protocolos informais contêm uma lista de métodos que as classes podem optar ou não por implementá-los, o que é bastante útil para a implementação de *delegates* (delegadores, que são demonstrados mais adiante neste capítulo). Por exemplo, imagine um componente de rolagem de conteúdo (scroll), que tem eventos para cada mudança na posição do conteúdo, nível de zoom e estado da animação de rolagem. Com a utilização de um protocolo informal, diferentes classes podem lidar apenas com os eventos que lhe interessam, sem precisar se preocupar em prover implementações vazias dos outros métodos.

Em contrapartida, *protocolos formais* definem um conjunto de métodos que devem obrigatoriamente ser implementados, o que é verificado pelo compilador. Um exemplo de uso deste tipo de protocolo é uma classe responsável por realizar download de arquivos, que determina que um método `downloadFinalizado` seja obrigatoriamente implementado, caso contrário não seria possível disponibilizar às classes o arquivo baixado.

Não existe uma regra sobre quando utilizar protocolos formais ou informais, sendo possível inclusive combiná-los em um único protocolo misto.

Você pode tentar definir o protocolo em um arquivo de cabeçalho (extensão `.h`) próprio, como em conjunto com a definição de uma classe. No caso de utilizar um arquivo de cabeçalho próprio, não é necessário a criação do arquivo `.m` — afinal, o protocolo define apenas o contrato.

A sintaxe é a seguinte:

```
@protocol EventosDeDownload <NSObject>
-(void) downloadFinalizado:(NSData *) conteudo;
-(void) downloadErro:(NSError *) erro;
@end
```

Por padrão todos os métodos de um protocolo são de implementação obrigatória. Caso você queira que alguns (ou todos) sejam opcionais, utilize a palavra-chave `@optional`:

```
// Arquivo "EventosDeDownload.h"
@protocol EventosDeDownload <NSObject>
-(void) downloadFinalizado:(NSData *) conteudo;

// Todos os métodos definidos daqui para baixo serão opcionais
@optional
-(void) downloadErro:(NSError *) erro;
@end
```

Se você quiser ser explícito, pode utilizar a palavra-chave `@required` para marcar os métodos que devem ser obrigatoriamente implementados. Para implementar um protocolo, a sintaxe é a seguinte:

```
#import "EventosDeDownload.h"

@interface AlgumaClasse : NSObject<EventosDeDownload>

@end
```

Para implementar diversos protocolos, basta separá-los por vírgula: `@NSObject<EventosDeDownload, OutroProtocolo>`. Agora é só implementar os métodos na classe respeitando a assinatura do método conforme definido na declaração do protocolo:

```
// Arquivo AlgumaClasse.m
@implementation AlgumaClasse

-(void) downloadFinalizado:(NSData *) conteudo {

}

-(void) downloadErro:(NSError *) erro {

}

@end
```

Uma nota sobre <NSObject>

Na declaração do protocolo `EventosDeDownload` foi adicionada a instrução `<NSObject>`, o que literalmente significa que o protocolo em questão implementa o protocolo `NSObject` (não confundir com a *classe* `NSObject`). Embora isso não seja obrigatório, é uma prática sempre declarar desta forma, pois permite a utilização de métodos como `retain` e `release`, assim como outros métodos definidos em `NSObject`.

11.7 Trabalhando com categorias

Uma das funcionalidades mais poderosas e úteis do Objective-C são as *Categorias*, que permitem estender as funcionalidades de *qualquer* classe, assim como substituir métodos já existentes. Categorias geralmente são criadas em seus próprios arquivos, embora seja possível adicionar várias em um mesmo arquivo-fonte, pois o compilador do Objective-C não impõe regras desta linha (ao contrário de Java). Além de poder adicionar e substituir métodos de qualquer classe, esta funcionalidade da linguagem pode atuar também como uma maneira de organizar o projeto, distribuindo a implementação da classes em arquivos distintos, agrupados por responsabilidade — ao contrário de colocar tudo em um único arquivo gigantesco. Elas também são úteis para declarar métodos privados em alguma classe sua.

Qualquer que seja o seu motivo, Categorias são extremamente poderosas e simples de utilizar, e do ponto de vista do programa, elas funcionam como se pertencessem à classe original, podendo inclusive acessar qualquer propriedade e método, *inclusive os privados*. É possível adicionar métodos tanto para as suas próprias classes, quanto para as classes da própria API oficial da linguagem, e não é necessário ter o código-fonte da classe. Por via de regra, não é possível adicionar variáveis à classe, apenas métodos — contudo, existem alguma técnicas que permitem simular o comportamento de variáveis de instância, o que é igualmente útil.

C# e VisualBasic.NET têm um recurso similar chamado *Extension Methods*, embora não permitam o acesso a membros privados. Já a comunidade Ruby se refere a isto como *Monkey Patching* (que na verdade é uma definição do conceito, não sendo exclusivo de Ruby).

Como primeiro exemplo, vamos adicionar um método na classe `NSString` para inverter o conteúdo da string:

```
// Arquivo NSString+Customizacoes.h
@interface NSString (Customizacoes)
-(NSString *) inverter;
@end
```

Não existe uma regra para o nome dos arquivos, porém um padrão comumente utilizado é *<NomeDaClasse>+<NomeDaCategoria>.extensão*, que no nosso caso virou `NSString+Customizacoes.h`. Na linha 2, repare que declaramos a `@interface` com o mesmo nome da classe `NSString`, porém com uma diferença muito importante: a adição de `(Customizacoes)`, que é a maneira como as categorias são declaradas em Objective-C. Você pode utilizar qualquer string dentro dos colchetes, desde que use o mesmo valor no arquivo de implementação, conforme o código abaixo:

```
// Arquivo NSString+Customizacoes.m
@implementation NSString (Customizacoes)

-(NSString *) inverter {
    char tmp[self.length + 1];

    for (int i = self.length - 1; i >= 0; i--) {
        char c = [self characterAtIndex:i];
        tmp[self.length - i - 1] = c;
    }

    tmp[self.length] = '\0';
    return [NSString stringWithUTF8String:tmp];
}

@end
```

O código da listagem acima é autoexplicativo, apenas atente ao detalhe que a declaração da `@implementation` contém o nome da categoria (`(Customizacoes)`) exatamente da mesma forma como foi declarada no arquivo `.h`. O método `inverter` agora está acessível diretamente na classe `NSString`:

```
NSString *s = @"uma palavra";
NSLog(@"%@", [s inverter]); // Resultado: "arvalap amu"
```

Categorias também são muito úteis para tornar mais fáceis códigos repetitivos. Um bom exemplo disso é a manipulação das propriedades de tamanho e posição

de uma `UIView`. Considere por um instante o código abaixo, que não faz uso de categorias:

```
1 // Obtêm a posição atual da view
2 float x = self.view.frame.origin.x;
3
4 // Diminui pela metade a posição
5 CGRect frame = self.view.frame;
6 frame.origin.x = x / 2;
7 self.view.frame = frame;
```

Embora o código da linha 2 não seja complicado, e ocupe apenas uma linha de código, ainda assim é bastante chato ter que passar por diversas propriedades até obter a posição "x" da view. Já o código das linhas 5 a 7 já é muito mais entendiante, e tê-lo que repetir diversas vezes durante o ciclo do programa é algo frustrante. Com categorias podemos simplificar bastante o processo:

```
// Arquivo UIView+AdicoesFrame.h
@interface UIView (AdicoesFrame)

-(void) setX:(float) newX;

// Em Objective-C evita-se o padrão "getX" de outras linguagens
-(float) x;

@end

// Arquivo UIView+AdicoesFrame.m
@implementation UIView (AdicoesFrame)

-(void) setX:(float) novoX {
    CGRect frame = self.frame;
    frame.origin.x = novoX;
    self.frame = frame;
}

-(float) x {
    return self.frame.origin.x;
}

@end
```

Tendo feito este código, a manipulação da view fica muito mais simples:

```
UIView *v = [[UIView alloc] init];
v.x = 45;

float posicaoX = v.x;
```

Você pode estender a mesma abordagem para outras propriedades, como `y`, `width` e `height`.

11.8 Gerenciamento de memória

Todo aplicativo, independentemente da linguagem em que foi desenvolvido e da plataforma que roda, precisa lidar com gerenciamento de memória — o que muda, porém, é quem é responsável por esta tarefa. Em Java e C# isso vem "de graça" na forma de um coletor de lixo automático, assim como Ruby. Por outro lado, C, C++ deixam esta tarefa para o próprio desenvolvedor. Já com Objective-C a resposta é "depende", pois existem dois conceitos: gerenciamento manual, e gerenciamento automático.

Toda aplicação que for criada a partir do Xcode 5 automaticamente terá habilitada por padrão o suporte a gerenciamento automático de memória, do termo *ARC* - "*Automatic Reference Counting*" (algo como "contador automático de referências"). Fazer uso de ARC é a maneira recomendada de construir aplicações modernas, havendo raríssimos motivos para não utilizá-lo. Não é necessário nenhuma configuração especial no seu projeto, o próprio Xcode já prepara o ambiente com suporte a ARC.

Objective-C em iOS versus Mac versus contador de referências

Em Mac existe o conceito de coletor de lixo (do inglês "*garbage collector*"), porém isso ainda não é realidade em iOS. Com o suporte a ARC no iOS, o compilador se encarrega de colocar as chamadas aos métodos de liberação de memória nos devidos lugares. Embora não chegue a ser um coletor de lixo propriamente dito, é uma grande mão na roda. Para utilizar esse recurso é necessário realizar algumas configurações especiais no projeto, além de tomar alguns cuidados na hora de desenvolver.

Apesar de parecer assustador, a realidade é muito mais tranquila, bastando seguir algumas receitas de bolo bastante simples para que tudo funcione em harmonia.

Contando referências

Objective-C trabalha com o conceito de contador de referências, em que, ao instanciar o objeto e toda vez que alguém o retém, um contador interno é incrementado. E quando uma mensagem para que seja liberado é executada, o contador é decrementado. Quando ele chegar em zero, a memória é liberada. A figura 11.1 exemplifica este conceito.

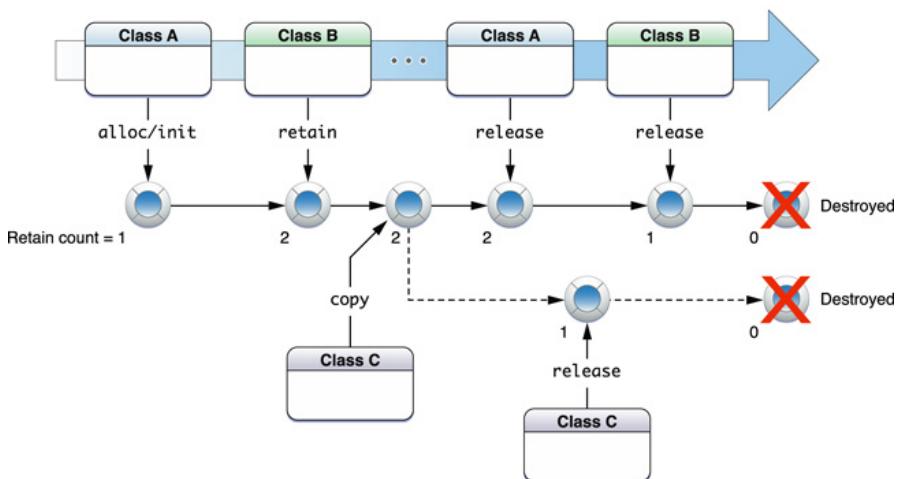



Figura 11.1: Exemplificação do contador de referências. Fonte: Apple

Como ARC é a maneira recomendada e padrão de criar projetos modernos para iOS, isso é tudo o que você precisa saber sobre gerenciamento de memória. Confie no compilador e ele fará todo o trabalho pesado para você. Caso esteja curioso, a imagem 11.2 mostra onde habilitar ou desabilitar o suporte a ARC.

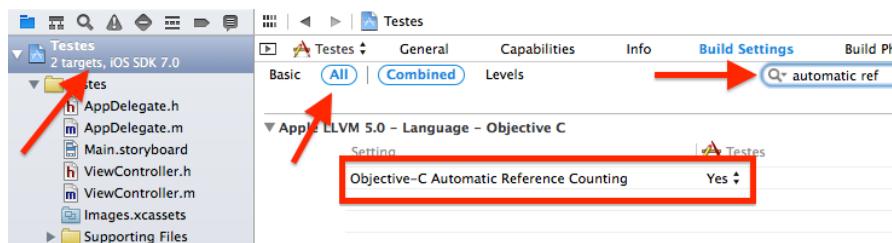



Figura 11.2: Chave de configuração para ARC - 'Yes' é o padrão

11.9 Gerenciamento manual de memória (para os curiosos)

Embora você sempre deva desenvolvedor seus projetos utilizando ARC (que é o padrão), esta seção irá descrever como funciona o gerenciamento manual de memória em Objective-C, tanto por razões históricas quanto para você compreender melhor o que se passa nos bastidores.

Caso queira, pode pular para a próxima seção sem medo. Na prática são bem remotas as chances de você ter que manualmente fazer o gerenciamento de memória.

Decidiu continuar a ler? Então vamos lá. Tudo se resume a três regras bastante simples e um pouco de atenção do desenvolver, nada mais. E, se algo der errado, o máximo que acontecerá é o seu aplicativo quebrar imediatamente, ou vazar memória até que eventualmente quebre.

As regras são:

- Se você é o dono, libere;
- Se você *não* é o dono, *não* libere;
- Sobrescreva o método `dealloc` em suas classes, para liberar as variáveis que possuir.

Primeira regra: quando você é o dono

Você é considerado "dono" de algum objeto quando utiliza explicitamente os métodos `alloc`, `copy`, `new` ou `retain`. Exemplo:

```
1 NSArray *dados = [[NSarray alloc] init];
2
```

```
3 // Trabalha com o NSArray
4
5 // Regra: somos donos, então liberamos
6 [dados release];
```

Na linha 1 utilizamos `alloc`, portando precisamos liberar a nossa parte, através do método `release` na linha 3. A respeito de `retain`, veja a regra 3.

Segunda regra: Se você não é o dono, não libere

Basta seguir a lógica inversa da primeira regra. Se você tem ou cria algo, mas não usou `alloc`, `new`, `copy` ou `retain`, deixe para lá. Por exemplo:

```
1 NSMutableDictionary *config = [NSMutableDictionary dictionary];
2 [config setObject:@"true" forKey:@"acesso.seguro"];
3 [config setObject:@"superSenha2000" forKey:@"senha.db"]
4
5 // Não é necessário fazer nada para liberar
```

Quando construímos o `NSMutableDictionary` na linha 1 da listagem anterior, não existe nada de `alloc` nem nenhum de seus parentes, portando não precisamos nos preocupar em liberar a memória manualmente — magia negra (ou alguma coisa assim) resolve o problema.

Terceira regra: Libere memória no método `dealloc`

Enquanto que em variáveis locais a memória também deve ser liberada no escopo do bloco de código, em variáveis de instância e nas marcadas com `@property` a liberação deve ocorrer num método especial chamado `dealloc`, o qual você deve sobrescrever. Para simplificar a explicação, considere uma hipotética classe `CarrinhoCompras`, a qual pertence a um `Cliente` e pode conter diversos `itens`:

```
1 // CarrinhoCompras.h
2 @interface CarrinhoCompras : NSObject {
3     NSMutableDictionary *cache; // Algum cache para operacoes internas
4 }
5
6 @property (nonatomic, retain) Cliente *cliente;
7 @property (nonatomic, readonly) NSMutableArray *itens;
8 @property (nonatomic, assign) BOOL estaPago;
9
```

```
10 @end
11
12 // CarrinhoCompras.m
13 @implementation CarrinhoCompras
14
15 -(id) init {
16     if ((self = [super init])) {
17         _itens = [[NSMutableArray array] retain];
18         cache = [[NSMutableDictionary alloc] init];
19     }
20
21     return self;
22 }
23
24 -(void) dealloc {
25     [_cliente release];
26     [cache release];
27     [_itens release];
28     [super dealloc];
29 }
30 @end
```

Para sabermos quais membros da classe precisam de gerenciamento manual de memória basta aplicar as regras do item 1, com a diferença que as chamadas a `release` devem ser feitas no método `dealloc`, conforme as linhas 25 a 28. Note que, nas linhas 17 e 18 iniciamos as variáveis `itens` e `cache` com `retain` e `alloc` respectivamente, e ambas se encaixam na primeira das regras que vimos. Contudo, o membro `cliente` está declarado de uma maneira ligeiramente diferente, através do uso de `@property` com o modificador `retain` - neste caso, o Objective-C se encarrega de parte do trabalho para nós, porém ainda assim é necessário o código da linha 25, caso contrário o contador de referências não seria decrementado. Se, por outro lado, usássemos `assign`, então *não* poderíamos invocar `release`.

Quando dealloc é executado? `dealloc` é executado automaticamente *quando for necessário. Contudo, você nunca pode chamar* `dealloc` *diretamente, exceto no caso de* `[super dealloc];` *(linha 28), que é* obrigatório. Lembre-se: use `release` para informar que você está liberando o objeto, e não `dealloc`.

ARC precisa estar desabilitado

Para usar retain e release o suporte a ARC precisa estar desabilitado. No mesmo lugar mostrado mais acima na imagem 11.2, mude a opção para "NO". Contudo, tenha em mente que cada vez mais bibliotecas de terceiros estão sendo atualizadas para utilizar ARC também, o que resultaria em diversos erros de compilação caso esta configuração seja desabilitada.

Na prática, é muito difícil conseguir combinar código ARC com código não ARC - é possível, mas igualmente fácil de dar algum tipo de problema.

11.10 Simplificando as coisas com literais

Uma das funcionalidades introduzidas no Xcode 4.5 é o conceito de literais (do inglês *Literals*), que é uma maneira bem mais simples e prática de iniciar coleções de objetos NSArray e NSDictionary, além da construção com menos código de objetos NSNumber.

Literais para NSNumber

A classe NSNumber é muito utilizada para encapsular tipos primitivos (char, short, int, long, long long, float, double, BOOL e bool) para que possam ser passados como argumentos para métodos que esperam objetos. Historicamente isso tinha que ser feito da seguinte maneira:

```
// Encapsulando tipos primitivos em um NSNumber da maneira tradicional,
// existente até o Xcode 4.4
NSNumber *letra = [NSNumber numberWithChar:'A'];
NSNumber *inteiro = [NSNumber numberWithInt:42];
NSNumber *yes = [NSNumber numberWithBool:YES];
```

Já com o Xcode 4.5 o mesmo código pode ser escrito da seguinte forma:

```
// Mesmo código feito no Xcode 4.5 em diante
NSNumber *letra = @'A';
NSNumber *inteiro = @42;
NSNumber *yes = @YES;
```

A diferença está em adicionar o caracter ‘ `@`’ na frente do valor a ser atribuído.

Literais para `NSArray`

As coisas ficaram bem mais simples para criar instâncias de `NSArray` também, pois até o Xcode 4.4 a inicialização somente poderia ser feita da seguinte forma:

```
// Criando NSArrays até o Xcode 4.4
NSArray *frutas = [NSArray arrayWithObjects:@"laranja", @"maçã",
    @"limão", @"melão", @"morango", nil];

NSString *morango = [frutas objectAtIndex:4];

NSArray *codigos = [NSArray arrayWithObjects:
    [NSNumber numberWithInt:33],
    [NSNumber numberWithInt:4345],
    [NSNumber numberWithInt:999],
    [NSNumber numberWithInt:17],
    nil];

int dezessete = [[codigos objectAtIndex:3] intValue];
```

Já a partir do Xcode 4.5 o mesmo código pode ser feito de maneira muito mais sucinta:

```
// Mesma construção, no Xcode 4.5 em diante
NSArray *frutas = @[@"laranja", @"maçã",
    @"limão", @"melão", @"morango"];
NSString *morango = frutas[4];

NSArray *codigos = @[@33, @4345, @99, @17];
int dezessete = codigos[3];
```

A regra é a mesma do `NSNumber`, bastando prefixar os valores com o caracter ‘ `@`’. Além disso, para acessar um determinado item pelo índice basta colocar o valor entre colchetes, ao invés de utilizar o método `objectAtIndex`. Um detalhe importante é que esta forma de inicialização irá criar apenas objetos imutáveis. Para criar instância de um `NSMutableArray` (que permite adicionar ou remover objetos a qualquer momento) faça da seguinte forma:

```
NSMutableArray *codigos =
    [NSMutableArray arrayWithArray:@[@45, @98, @1234]];
```

```
[codigos addObject:@[@897]];
[codigos addObject:@[@333]];
```

[Literais para NSDictionary] Outra classe que tira grande proveito de literais é a NSDictionary, pois além dos valores em si é necessário especificar as chaves para cada um dos valores, o que torna o código um tanto chato de ler, conforme mostrado abaixo:

```
// Criando e acessando NSDictionary até o Xcode 4.4
NSDictionary *dados = [NSDictionary
    dictionaryWithObjectsAndKeys:@"valor1", @"chave1"
    @"valor2", @"chave2",
    @"valor3", @"chave3",
    @"valor4", @"chave4",
      nil];

NSString *valorChave2 = [dados objectForKey:@"chave2"];
```

Utilizando os literais a partir do Xcode 4.5, o mesmo código pode ser escrito da seguinte forma:

```
// Versão Xcode 4.5 em diante
NSDictionary *dados = @{
    @"chave1" : @"valor1",
    @"chave2" : @"valor2",
    @"chave3" : @"valor3",
    @"chave4" : @"valor4",
};

NSString *valorChave2 = dados[@"chave2"];
```

A inicialização foi simplificada com o uso de `@{}`, e a separação entre chave e valor é feita com o caractere dois-pontos (":"). De maneira similar ao NSArray, uma determinada chave pode ser acessada utilizando colchetes (como `dados[@"chave2"]`) ao invés do método `objectForKey:`.

Capítulo 12

Como criar uma conta no portal de desenvolvimento da Apple

A Apple disponibiliza, através do *iOS Developer Program*, a suíte completa de ferramentas, documentação e códigos de exemplo para você desenvolver e distribui aplicativos para iOS. Para começar a criar aplicativos tudo o que você precisa fazer é criar uma conta grátis no site de desenvolvedores.

12.1 Registre-se como um desenvolvedor Apple

O primeiro passo é se registrar como um desenvolvedor Apple, o que pode ser feito de duas maneiras distintas: criando um *Apple ID* do zero, ou então utilizar o seu login existente do iTunes (o mesmo utilizado para baixar aplicativos da App Store, caso você tenha um iPhone, iPod Touch ou iPad). O resultado final de ambos os processos é o mesmo.

Se você já tem um Apple ID basta acessar o endereço http://developer.apple.com/

devcenter/ios, clicar no botão *Log in* e fornecer as mesmas credenciais utilizadas para baixar aplicativos na App Store. Simples assim.

Caso você ainda não tenha um Apple ID acesse o link https://developer.apple.com/programs/register e clique no botão *Create Apple ID*. Note que todas as informações devem ser inseridas em inglês, evitando caracteres acentuados. Informe dados como e-mail, senha, endereço e o que mais for solicitado. Se não souber o que preencher no campo *Company / Organization*, informe *Private*. No final do processo a Apple enviará um email para o endereço informado; abra-o e clique no link de verificação.

Depois que você criou a sua conta ou entrou com uma existe, será solicitado algumas informações gerais sobre você, como perfil profissional e para quais plataformas Apple presente desenvolver, conforme mostra a figura 12.1. Obs: caso você utilize uma conta Apple criada anteriormente, pode acontecer de não aparecer a tela da imagem 12.1, e sim cair direto no "iOS Dev Center". Neste caso as informações devem ter sido inseridos em algum outro momento.

Figura 12.1: Primeira parte do processo de criação do Apple ID

12.2 Fazendo a assinatura no iOS Developer Program

Acesse o endereço https://developer.apple.com/devcenter/ios para ter acesso a todos os downloads, documentação e códigos de exemplo.

Enquanto que o acesso ao portal de desenvolvimento, juntamente com as ferramentas e documentação são disponibilizadas gratuitamente pela Apple, para rodar o aplicativo em dispositivos e distribuí-lo na App Store é necessário fazer a assinatura do *iOS Developer Program*, independentemente se o seu aplicativo será gratuito ou pago. A assinatura deste programa de desenvolvimento é anual, e custa US$ 99,00 tanto para pessoas físicas quanto para empresas.

> **Assinatura versus conta grátis**
>
> Para começar a desenvolver aplicativos para iOS não é necessário realizar a assinatura do iOS Developer Program, bastando apenas o cadastro simples explicado anteriormente neste capítulo, que permite rodar os aplicativos no simulador que é instalado com o Xcode.

Para realizar a assinatura, acesse o endereço https://developer.apple.com/programs/ios e clique no botão *Enroll Now*. O processo todo é feito da seguinte forma:

Fornecer os dados de acesso

A primeira parte, identificada como *Sign in or create an Apple ID*, consiste em informar se você deseja realizar o cadastro com um Apple ID existente ou criar um novo. Considerando que você já tem o Apple ID, selecione a opção *Sign in with your Apple ID* e continue. Obs: caso já esteja logado no portal de desenvolvimento, você verá a opção "Existing Apple ID" com um botão "Continue". Clique nele se for o seu caso.

Assinar uma conta individual ou como empresa

É possível realizar a assinatura como pessoa física (conta individual), ou como empresa, sendo que a principal diferença entre elas é que uma conta individual permite que apenas um desenvolvedor (você) tenha permissão para acessar o portal de

desenvolvimento, enquanto que a conta empresarial permite criar times de desenvolvedores. Selecione a opção *Individual* no fim da página, e forneça o seu Apple ID e senha caso solicitado.

Informações pessoais para validação de segurança

Após realizado o login irá aparecer uma tela com o título *Enter your billing information for identity verification*, conforme mostrado pela figura 12.2. Na primeira parte, chamada *Your Name*, informe o seu nome **exatamente** como aparece no seu cartão de crédito internacional que será utilizado para realizar o pagamento da assinatura mais adiante, e em *Your credit card billing address* informe o endereço de correspondência da fatura do cartão de crédito. Clique em *Continue*.

Figura 12.2: Informações pessoais

Selecionar o tipo de assinatura

A próxima tela, demonstrada na figura 12.3, pede para informarmos qual assinatura desejamos fazer. Selecione *iOS Developer Program (US$99 /year)* e clique no botão "Continue".

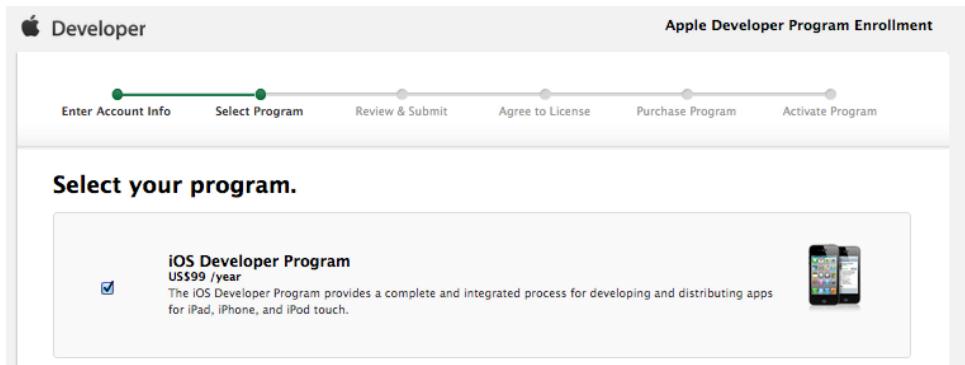



Figura 12.3: Selecionar o programa

Revisar as informações

A próxima tela, *Review your enrollment information & submit*, pede apenas para você verificar se todas as informações fornecidas até o momento estão corretas. Pode continuar.

Contrato padrão

A tela *Program License Agreement* pede para você ler e aceitar o contrato padrão que é apresentado. Selecione o checkbox confirmando o aceite e clique no botão *I Agree*.

Dados do cartão de crédito

Na tela "Enter your payment information" você deve inserir os dados do seu cartão de crédito internacional, como em qualquer outra loja de ecommerce. Preencha as informações solicitadas e clique em "Continue" no final da página. A próxima tela lhe apresentará um resumo de todas as informações inseridas até agora, revise se está tudo OK e clique no botão "Place Order", conforme a imagem 12.4.

Member Information

Name: Rafael Steil
Email: rafael@example.com
Enrollment ID: G2JKLEV36D87 Guarde este código

Cancel Place Order

Figura 12.4: Código de identificação da sua conta, guarde-o muito bem

Guarde muito bem o Enrollment ID

Atenção: guarde muito bem o código do *Enrollment ID*, pois eles serão necessários para a renovação da assinatura nos próximos anos. A Apple não disponibiliza em nenhum outro lugar este código, porém solicita-o no processo de renovação.

Ao final, você será apresentado com uma tela como a da imagem 12.5. A Apple diz levar até dois dias úteis para processar os eu cartão de crédito, porém é bastante comum eles errarem ou simplesmente esquecerem o seu pedido, por mais absurdo que isso possa parece. Se passar muito tempo do prazo e você não tiver respostas, ligue no suporte técnico da Apple nos Estados Unidos (infelizmente não há canal direto de atendimento no Brasil) e relate o ocorrido. Já aconteceu da Apple cobrar mais de uma vez o mesmo pedido, e frequentemente passam dias sem dar qualquer informação, portanto o melhor canal é via telefone.

Figura 12.5: Tela final

12.3 Os tipos de certificados

Todo aplicativo que roda no iOS precisa ser assinado digitalmente com a chave do desenvolvedor e uma contra-chave da Apple, que juntas garantem a legitimidade do seu aplicativo. Existem certificados para desenvolvimento, homologação e para produção, chamados nos termos em inglês de *Development*, *Ad-hoc* e *Distribution* respectivamente. Os certificados de desenvolvimento e produção você utilizará com certeza, enquanto que o de homologação (*Ad-hoc*) somente é necessário se desejarmos enviar o aplicativo para outras pessoas testá-lo antes de enviar oficialmente para a App Store. Porém, como é explicado neste capítulo, existe um limite anual para dispositivos de homologação.

Para testar os aplicativos durante o processo de desenvolvimento, é possível utilizar o simulador de iOS que é instalado juntamente com o Xcode, o qual não precisa de um certificado específico, e rodá-lo no seu iPhone ou iPad através do certificado de desenvolvimento.

Capítulo 13

Rodando os aplicativos no seu iDispositivo

Até agora todos os aplicativos que fizemos no livro rodaram no simulador do iOS que vem junto com o Xcode, e neste capítulo iremos ver os passos necessários para rodar os mesmos aplicativos diretamente no dispositivo, seja ele um iPhone, iPad ou iPod Touch ou qualquer outro iGadget que posteriormente a Apple venha a lançar. A razão de dedicarmos um capítulo inteiro para este assunto é que, ao contrário do simulador, os iDevices somente podem rodar aplicativos assinados digitalmente, o que requer tanto um certificado do desenvolvedor quanto um certificado da Apple, num processo chamado "provisionamento", do termo em inglês "*provisioning*"

Existem três tipos de provisionamento: para desenvolvimento, para testes e homologação, e para distribuição na App Store. O de desenvolvimento permite rodar e debugar o aplicativo no dispositivo através de um cabo USB, diretamente pelo Xcode. O para testes e homologação, chamado de "Ad-hoc", é útil para distribuir o aplicativo para que outras pessoas possam ajudar a testar ou validar — o caso mais comum é o

cliente para o qual você está criando a app. Já o de distribuição (do inglês "*distribution*") é utilizado para enviar a versão final para a App Store de fato.

Embora focaremos no provisionamento de desenvolvimento, os passos necessários para o de Ad-Doc e Distribuição são virtualmente os mesmos.

Utilize o Safari caso tenha problemas em alguns passos

Por algum motivo inexplicável e sem razão alguma, é necessário realizar os passos deste capítulo utilizando o Safari ao invés de qualquer outro browser, pois muito frequentemente o sistema online da Apple irá acusar que os arquivos enviados via upload estão corrompidos. Utilizando o Safari este problema não ocorre.

Embora você *talvez* consiga realizar os procedimentos com sucesso com outro browser, o mais seguro mesmo é não abusar da sorte, e ir pelo Safari.

Todos os passos descritos neste capítulo assumem que você já tenha a conta paga de desenvolvedor.

13.1 Crie e instale o certificado

O primeiro passo é obter o certificado digital que será utilizado para criar as chaves de provisionamento. Acesse o *iOS Provisioning Portal* no endereço https://developer.apple.com/account/overview.action, clique no link *Certificates* no menu esquerdo, e depois no botão com o símbolo de soma, conforme a figura 13.1. Na tela que se abrir, selecione "*iOS App Development*" e clique em "*Continue*".

Figura 13.1: Primeiro passo para solicitar o certificado digital

A tela seguinte deverá ser uma com o título "*About Creating a Certificate Signing Request (CSR)*", que contêm apenas uma explicação geral. Clique mais uma vez em "*Continue*".

Para criar o certificado primeiro precisamos enviar um arquivo de solicitação para a Apple, chamado de *Certificate Signing Request (CSR)*, o qual contém uma chave de criptografia pública, que a Apple utilizará para emitir o certificado "de verdade". Ele é o que ficará vinculado à sua conta de desenvolvedor para todos os propósitos. No seu Mac, abra o aplicativo *Keychain Access* (localizado em *Applications -> Utilities*), e acesse o menu *Keychain Access -> Certificate Assistant -> Request a Certificate from a Certificate Authority* conforme a figura 13.2

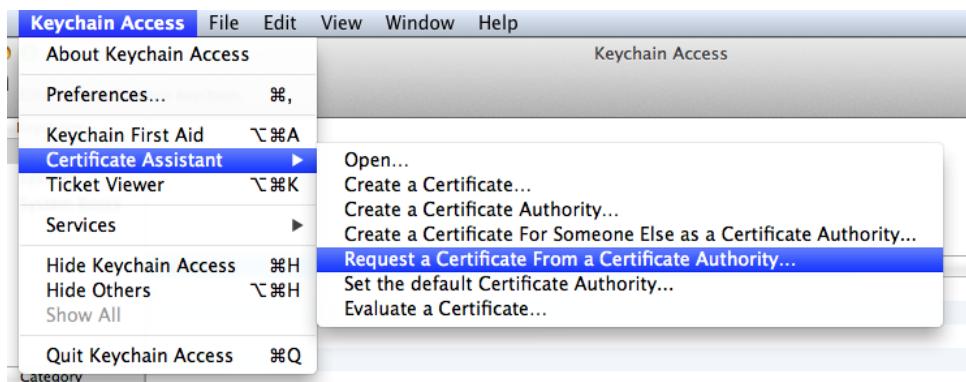



Figura 13.2: Menu do Keychain para acessar

Na tela que abrir, informe o seu endereço de e-mail no campo *User email*

address, e selecione a opção *Saved to disk*, conforme a figura 13.3. Depois clique no botão *Continue*, e o Keychain irá pedir para você salvar o arquivo `CertificateSigningRequest.certSigningRequest` em algum lugar. Este é o arquivo que deverá ser enviado para o portal de desenvolvimento.

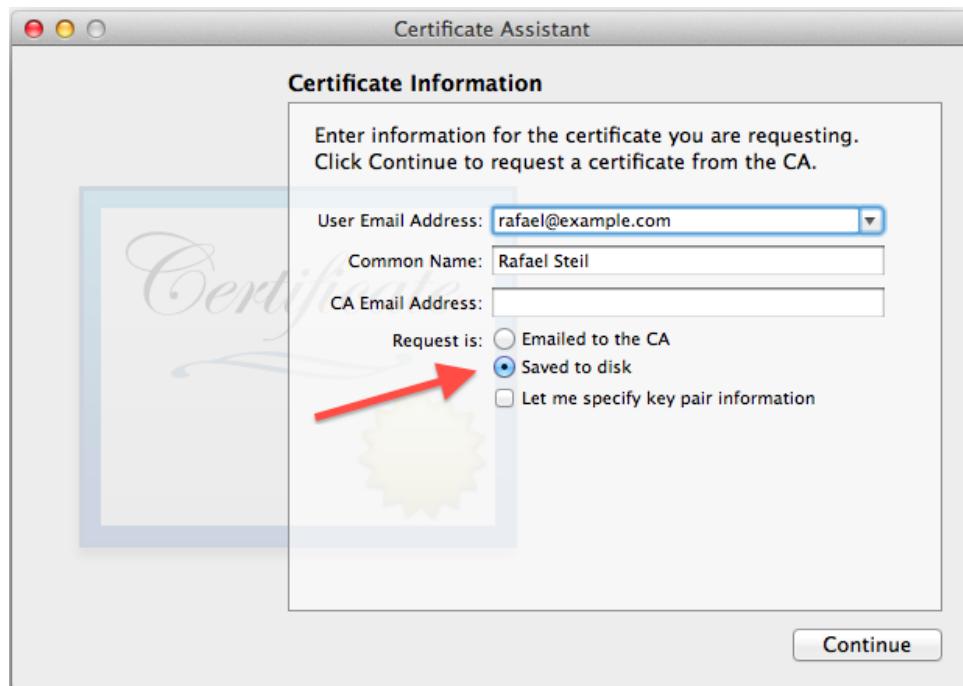



Figura 13.3: Primeira tela de criação da chave pública

Após salvar o arquivo `CertificateSigningRequest.certSigningRequest` em disco, volte para a tela *Certificates* do portal de desenvolvimento, selecione o arquivo no botão *Choose File*, e clique no botão *Submit*, conforme a figura 13.4. Lembre-se de fazer este passo pelo Safari caso seja informada alguma mensagem de erro ao enviar.

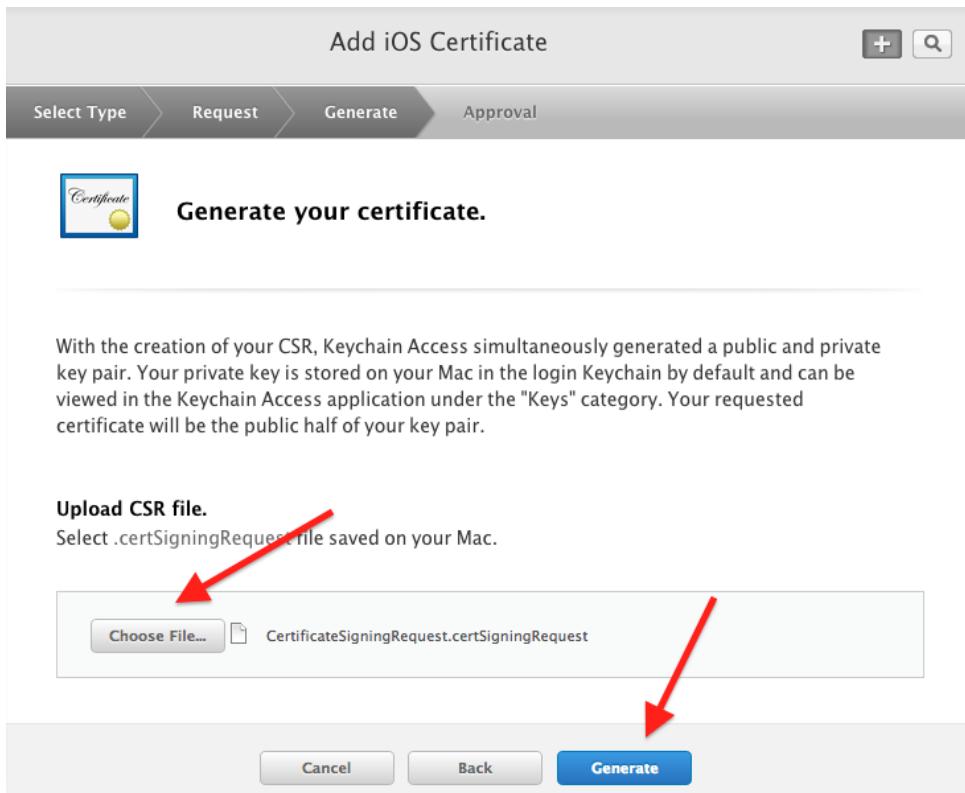



Figura 13.4: Enviando o CSR para a Apple

Se tudo ocorrer sem problemas, o browser será redirecionado de volta para a tela de certificados, Se estiver aparecendo apenas a mensagem *Pending Issuance*, espere alguns minutos e dê refresh na tela, pois às vezes o sistema demora um pouco para liberar o download do certificado.

Na tela da figura 13.5 clique em cima do nome do certificado para expandir as opções, e clique no botão "*Download*". Para instalá-los basta dar um duplo clique no arquivo, e ele será importado para dentro do *Keychain*, conforme a figura 13.5. Note que deverá aparecer o "*Apple Worldwide Developer Relations Certification Authority*", e o seu "*iPhone Developer*" do tipo *Certificate* e *Private Key* - se houver apenas algum deles (geralmente somente o *Certificate*), algum passo foi realizado incorretamente. Obs: o "*Apple Worldwide Developer Relations Certification Authority*" é instalado automaticamente pelo Xcode, portanto você não deve ter problemas.

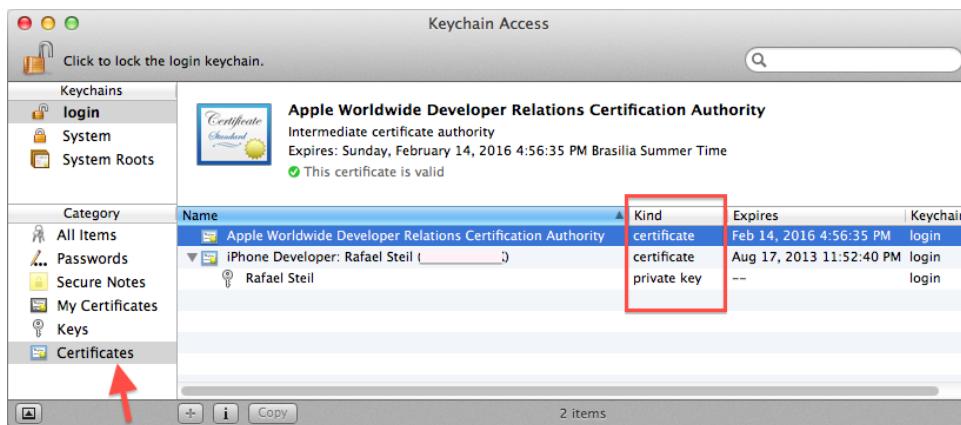



Figura 13.5: Certificados instalados com sucesso

Certificados de desenvolvimento e distribuição

É importante frisar que existem dois certificados que você irá eventualmente precisar: o primeiro é o de Desenvolvimento (*Development*), que acabamos de criar, e serve para a fase de desenvolvimento do aplicativo. O outro é o certificado de Distribuição (*Distribution*), utilizado para enviar para clientes homologarem antes de entrar na App Store e para enviar para a App Store de vez, para o público final. O procedimento para criar o certificado de Distribuição é o mesmo do de Desenvolvimento, apenas devendo ser feito na aba "*Distribution*" do menu "*Certificates*".

13.2 Crie a identidade do seu aplicativo - App IDs

Uma parte vital do seu aplicativo é sua identificação, chamada de *App ID*, a qual é utilizada em todas as partes do ecossistema do iOS, inclusive para distribuição na App Store. O App ID é um identificador único, sendo que existe a possibilidade de criar um coringa para a sua conta, para que possa ser utilizado em diversos aplicativos (com restrições, porém).

Para criar um novo App ID selecione a opção *App IDs* no menu lateral e em seguida clique no botão representado pelo sinal de soma, que abrirá a página da figura 13.6. O primeiro campo (*App ID Description*) é uma breve descrição do aplicativo

apenas para o seu próprio controle interno, e no segundo campo do formulário (*App ID Prefix*) selecione a opção *Team ID*. O terceiro campo (*Bundle ID*) será o identificador em si, que por via de regra deve ser único dentre *todos* aplicativos existentes na App Store. A recomendação da própria Apple é utilizar um estilo de nomenclatura que se assemelhe a um domínio de internet reverso seguido do nome do aplicativo em si, como `br.com.example.MeuSuperApp`. Pessoalmente vou um passo além e coloco como sufixo a plataforma (iPhone ou iPad) caso exista a possibilidade de lançar versões diferentes do aplicativo para iPhone e iPad — neste caso, o App ID ficaria sendo algo como `com.rafaelsteil.MeuSuperApp.iphone` e `com.rafaelsteil.MeuSuperApp.ipad`.

Figura 13.6: Formulário de criação de um novo App ID

Após preencher o formulário e clicar no botão *Submit* você deverá voltar para a

tela de listagem anterior, na qual estará listado o aplicativo. Se você clicar em cima do nome dele, um box irá expandir mostrando o que está habilitado. Clique no botão *Edit: para mostrar a tela onde é possível habilitar suporte a* Push Notifications, iCloud:: e outras funcionalidades opcionais. Por hora não é necessário modificar nada, e podemos prosseguir para o próximo passo.

App ID coringa

A Apple permite a criação de um App ID coringa, que pode ser utilizado para diversos aplicativos distintos sem que haja a necessidade de criar um ID especifico para cada um, o que é bastante vantajoso quando estamos criando vários aplicativos de testes. Para criar o App ID coringa basta selecionar a opção "*Wildcard App ID*" e inserir $\star$ (asterisco) no campo *Bundle ID*) da figura 13.6.

Somente é possível criar um App ID coringa por conta. A Apple utiliza o nome *Catch All* para se referenciar a este tipo de App ID.

13.3 Adicionando dispositivos para desenvolvimento

Para que seja possível rodar os seus aplicativos durante a fase de desenvolvimento, é necessário primeiro cadastrá-los no *iOS Development Portal*. O cadastro do dispositivo, que pode ser qualquer iDevice, é feito com o fornecimento de um ID único chamado *UDID*, do termo em inglês *Unique Device Identifier* (ou "Identificador único de dispositivo"), que é um código com 40 caracteres. Para localizá-lo, conecte o seu dispositivo ao Mac via cabo USB, abra o Xcode e acesse o menu *Window -> Organizer*, e selecione-o na seção *Devices* do menu esquerdo, conforme mostra a figura 13.7. O UDID é aquele código ao lado do texto "*Identifier*". Copie o código (repare que é um campo de texto selecionável).

Figura 13.7: Localizando o UDID do seu dispositivo

Repare que na figura 13.7 existe um botão chamado *Use for Development*, que habilita o dispositivo para ser utilizado para tarefas de desenvolvimento, como rodar diretamente do Xcode e utilizar os recursos de debug. Certifique-se de selecionar esta opção.

Tendo o UDID em mãos, acesse a opção *Devices* no portal e clique no botão com o símbolo de soma. No formulário que se abre coloque um nome descritivo do dispositivo no campo *Name*, e o UDID obtido anteriormente no campo *UDID*. Clique em *Submit* e ele deverá aparecer na listagem.

13.4 Limite anual de dispositivos

A Apple impõe um limite anual de 100 dispositivos para serem utilizados para testes e homologação, permitindo resetar o cadastro uma única vez por ano, após a renovação da conta de desenvolvimento. Este número de 100 dispositivos é sempre incremental durante o ano, nem mesmo diminuindo se você remover algum registro. Embora para o desenvolvedor independente esta restrição não chega a ser um grande problema, no caso de empresas que atendem a diversos clientes é um fator bastante preocupante, pois uma vez atingido o limite anual não há qualquer maneira de conseguir cadastrar mais dispositivos, sendo necessário esperar o período de re-

novação da conta.

Depois de renovar a conta do *iOS Developer Program*, a seção *Devices* do portal irá mostrar um aviso em amarelo informando que você tem agora a opção de remover "de verdade" os registros que deseja. **Faça isso antes de adicionar qualquer outro dispositivo**, pois ao adicionar um novo registro a sua conta ficará travada por mais um ano.

13.5 Crie o certificado de provisionamento

Agora que já temos a nossa chave privada, o certificado da Apple, o App ID e o dispositivo de testes devidamente registrado, o último passo é obter o arquivo de provisionamento (*Provisioning Profile*). Ele é utilizado para assinar digitalmente o aplicativo de tal forma que seja possível rodá-lo nos dispositivos cadastrados sem passar pela App Store. Para tanto, acesse o menu *Development* da seção "*Provisioning Profiles*" no menu lateral, e em seguida clique no botão com o símbolo de soma.

No formulário que abrir selecione "*iOS App Development*" e clique no botão "*Continue*", na próxima tela selecione o "*App ID*", na tela seguinte marque o certificado a utilizar (deverá haver um único). Depois de clicar em "*Continue*" mais uma vez, selecione o dispositivo na lista e clique novamente no botão de continuar. Por último, coloque um nome descritivo para o profile, e clique no botão "*Generate*".

Reconfigure o *provisioning* após adicionar um novo dispositivo

Lembre-se que toda vez que você cadastrar um novo dispositivo é necessário reconfigurar o arquivo de provisionamento, bastando para isso clicar no link *Edit* e depois em *Modify* na tela principal do link *Provisioning* do menu lateral (a mesma mostrada na figura 13.8), e por último marcar os dispositivos desejados.

Na listagem dos provisionamentos, clique em cima do nome do que foi criado no passo anterior, e em seguida no botão "*Download*".

Caso tudo tenha ocorrido com sucesso você deverá ver o arquivo de provisionamento listado na tela principal do menu *Provisioning*, e o botão *Download* habilitado, conforme mostra a figura 13.8. Faça o download deste arquivo (ele terá a extensão

.mobileprovision) e dê duplo clique para abri-lo no *Organizer* do Xcode. Após a importação, ele deverá estar listado com o status *Valid Profile*, na seção *Provisioning Profiles* da *Library* do organizer, conforme mostra a figura 13.8.

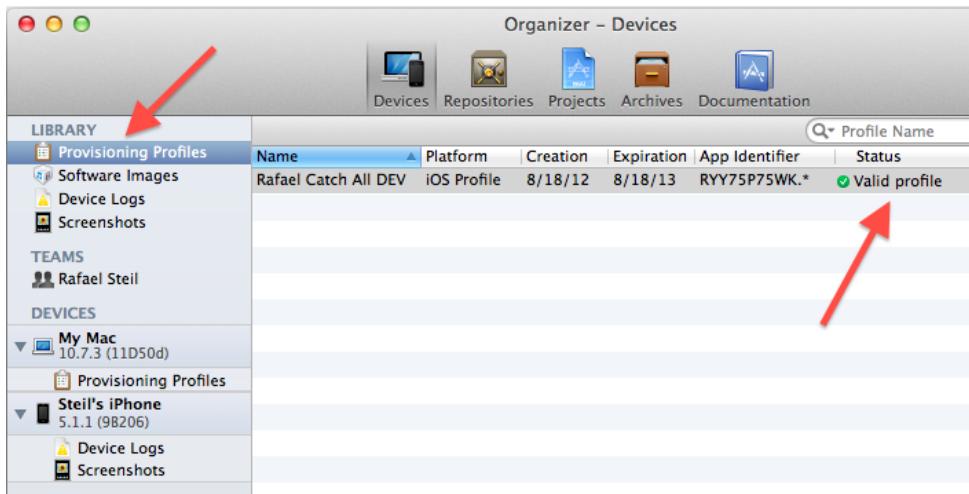



Figura 13.8: Provisionamento importado corretamente

13.6 Associe o arquivo de provisionamento no Xcode

O último passo para poder rodar os aplicativos nos dispositivos cadastrados é configurar o Xcode para utilizar o arquivo de provisionamento apropriado. Abra as propriedades do projeto desejado, selecione o *Target* apropriado (por padrão somente existirá um, correspondente ao nome do projeto) no menu lateral, e selecione a aba *Build Settings*. Nela, você deverá localizar a seção *Code Signing -> Code Signing Identity*, que deverá ter o valor ::Don't Code Sign", o qual iremos modificar. Veja a figura para referência 13.9.

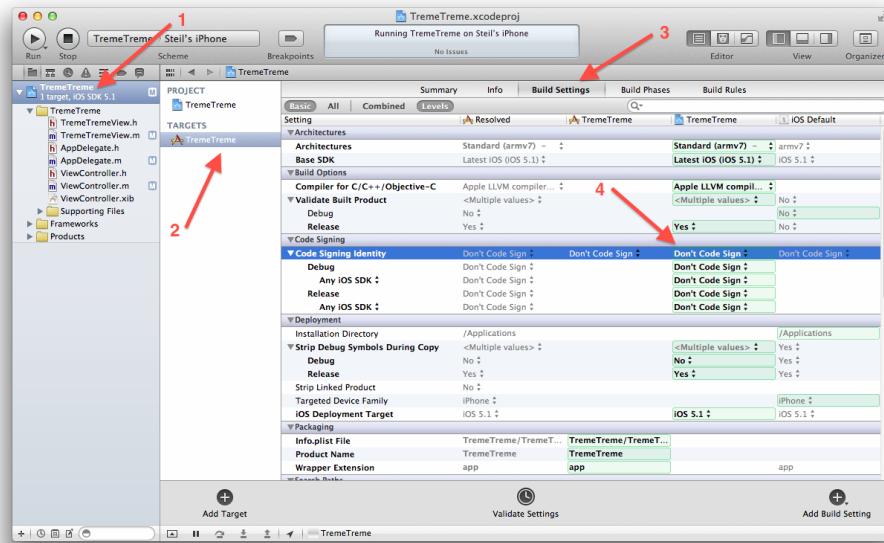



Figura 13.9: Localizando a seção para seleção do arquivo de provisionamento

Na linha do *Code Signing Identity* clique em cima de *Don't Code Sign* para abrir o menu de seleção de perfis de provisionamento, e selecione a opção *iPhone Developer*, conforme mostra a figura 13.10. A opção *iPhone Developer* se encarrega de automaticamente utilizar o perfil de provisionamento mais apropriado. Por via de regra, ele irá verificar o valor do campo *Bundle Identifier* na aba *Summary* e comparar com os perfis existentes, utilizando algum identificador que seja igual ao cadastrado na seção *App IDs* no portal da Apple (veja a figura 13.6). Caso contrário, irá utilizar o *Catch All* (coringa) caso exista.

Reforçando a importância do *Bundle Identifier*

Preste bastante atenção em relação ao fluxo que é descrito neste capítulo, pois uma vírgula errada poderá fazer você perder muitas horas de trabalho (e fios de cabelo) tentando encontrar o que deu errado. Isso é especialmente válido para o *Bundle Identifier*, o qual deve ser correspondente no Xcode ao valor cadastrado na seção *App IDs* no portal da Apple.

Por exemplo, se no terceiro campo da figura 13.6 você inseriu o valor "*com.exemplo.MeuSuperApp.iphone*", é exatamente este valor que deverá ser utilizado no campo *Bundle Identifier* no Xcode (*Propriedades do projeto -> Summary -> Bundle Identifier*).

Figura 13.10: Selecionando o perfil iPhone Developer

Uma vez selecionado o perfil, o resultado deverá ficar conforme o da figura 13.11

Figura 13.11: Perfil selecionado corretamente para todas as opções

13.7 Rode seu aplicativo no dispositivo

Agora que temos tudo configurado, para rodar o aplicativo no dispositivo basta conectá-lo ao Mac via cabo USB e selecioná-lo como destino no Xcode, conforme mostrado na figura 13.12. Compile o projeto (*Command* + *B*) e rode-o utilizando *Command* + *R*. Caso apareça uma caixa de diálogo pedindo permissão para utilizar a sua chave (aquela criada anteriormente, no início do capítulo), selecione a opção "*Always Allow*" ("Permitir sempre").

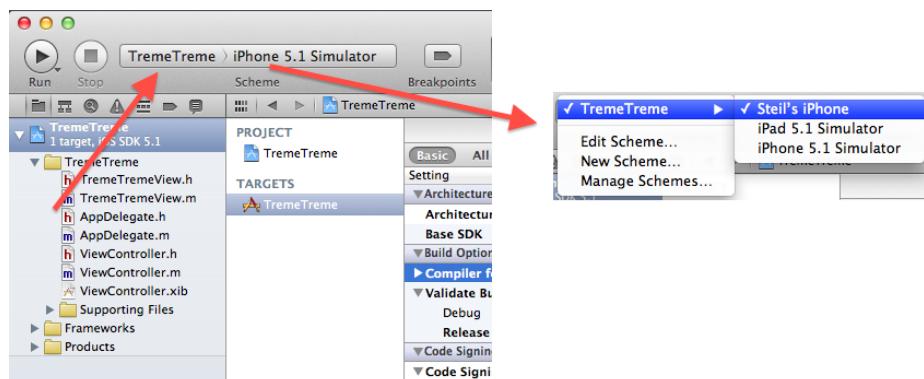



Figura 13.12: Selecionando o dispositivo para rodar o aplicativo

Caso o Xcode aponte algum problema como o da figura 13.13, verifique se o perfil *iPhone Developer* está selecionado em todas as linhas da seção *Code Signing Identity*, conforme a figura 13.11.

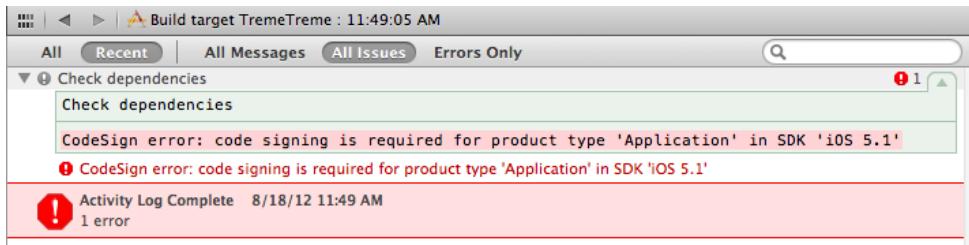



Figura 13.13: Falha ao compilar devido a erro na configuração do perfil de provisionamento

13.8 Verificando a instalação dos perfis no dispositivo

Para verificar os perfis instalados em um determinado dispositivo basta acessar a app *Settings*, ir na seção *General* e procurar pelos perfis no final da tela, conforme mostra a figura 13.14.

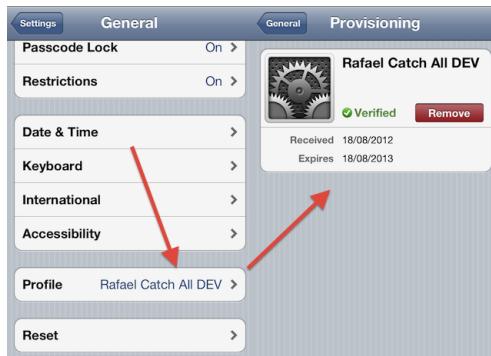



Figura 13.14: Localização do perfil de provisionamento no dispositivo

Capítulo 14

Uma palavra final + bônus

Uma das lições (vinda em forma de conselho) mais importantes que aprendi logo que comecei a trabalhar com desenvolvimento de software, e a qual de certa forma me moldou em diversos aspectos, foi a seguinte: o nosso trabalho é encontrar soluções, não reclamar que tudo é difícil e que "não dá para fazer", ficando sem ação. Se o problema está muito difícil, quebre-o em pedaços menores até que seja possível resolvê-lo. E se não dá para fazer da maneira solicitada, então encontre uma forma possível.

As empresas têm problemas que precisam ser solucionados, e para isso tentam contratar as melhores pessoas possíveis. Você, ao fazer parte da equipe, tem a responsabilidade de encontrar e desenvolver a solução. Se não fizer, quem fará? Pense nisso.

Abraço, Rafael

14.1 Bônus - livros e links

Conhecimento nunca é demais, e abaixo estão alguns livros e sites extremamente úteis.

- Learn Objective-C for Java Developers, ISBN 978-1430223696 (http://amzn.to/PwFyQl)
- Objective-C for Absolute Beginners, ISBN 978-1430236535 (http://amzn.to/Q3moTm)
- Stackoverflow (http://bit.ly/oW30Ca)
- Principal documentação da Apple sobre desenvolvimento iOS (http://bit.ly/dGP5Cm)
- Guia da Apple para desenvolvimento de aplicativos (http://bit.ly/PXZEW4)
- http://www.icodeblog.com (blog e artigos)
- http://iosdevelopertips.com (blog e artigos)
- http://maniacdev.com (blog e artigos)
- http://www.raywenderlich.com (blog bastante famoso, com artigos muito bem elaborados)
- http://www.cocoacontrols.com (muitos componentes e projetos Open Source)